Softwaretechnik

Prof. Dr. Rainer Koschke

Fachbereich Mathematik und Informatik
Arbeitsgruppe Softwaretechnik
Universität Bremen

Wintersemester 2013/14

Überblick I

Vorbemerkungen

Software-Metriken

Entwicklungsprozesse

Testgetriebene Entwicklung und Paarprogrammierung

Kosten- und Aufwandsschätzung

Empirische Softwaretechnik

Kontrollierte Experimente

Statistik bei kontrollierten Experimenten

Überblick II
Entwurfsmuster

Komponentenbasierte Entwicklung

OSGI als Beispiel eines Komponentenmodells

Modellgetriebene Softwareentwicklung

Demo zur Modellgetriebenen Softwareentwicklung

Software-Produktlinien

Vorbemerkungen

Vorbemerkungen
Themen der Vorlesung
Übersicht
Termine
Übungen und Ressourcen
Scheinbedingungen
Lehrbücher

Übersicht I

- Entwicklungsprozesse
- Metriken
- Empirie in der Softwaretechnik
- Kosten- und Aufwandsschätzung
- Komponentenbasierte Entwicklung
- Entwurfsmuster (Schwerpunkt Parallelität)
- Software-Architektur
- Modellgetriebene Softwareentwicklung
- Software-Produktlinien

Entwicklungsprozesse

- alternative Software-Entwicklungsprozesse (z.B. Clean-Room und Extreme Programming)
- Capability Maturity Model, Spice und Bootstrap
- Prozessverbesserungen
- Persönlicher Prozess
- Literatur: Balzert (2008); Bunse und von Knethen (2002); Kneuper (2006); Siviy u. a. (2007)

Softwaremetriken

- was ist eine Metrik?
- Messtheorie
- Skalen
- Prozess-, Produkt- und Ressourcenmetriken
- Literatur: Fenton und Pfleeger (1998)

Kosten- und Aufwandsschätzung

- insbesondere Function-Points und CoCoMo I und II
- Literatur: Poensgen und Bock (2005); Boehm u. a. (2000)

Komponentenbasierte Entwicklung

- Eigenschaften, Vor- und Nachteile
- Komponentenmodell
- Schnittstellen und Kontrakte
- Managementfragen
- Rahmenwerke
- existierende Komponentensysteme, z.B. .NET, Microsoft DCOM, OLE, ActiveX, Sun Java und JavaBeans
- Literatur: Szyperski u. a. (2002)

Modellgetriebene Softwareentwicklung

- Ideen, Eigenschaften, Vor- und Nachteile
- Werkzeugunterstützung am Beispiel von Eclipse Open Architecture Ware
- Literatur: Stahl u. a. (2007)

Software-Architektur

- Entwurfsmuster
- Qualitätseigenschaften
- Analyse von Architekturen (insbesondere SAAM und ATAM)
- Literatur: Buschmann u. a. (1996); Gamma u. a. (2003); Bass u. a. (2003); Hofmeister u. a. (2000)

Software-Produktlinien

- Definition und Beispiele
- Vor- und Nachteile
- Practice Areas
- Einführung von Produktlinien
- Ansätze zur technischen Realisierung
- Beschreibungen und Notationen (z.B. Feature-Graphen)
- Besonderheiten beim Requirementsengineering, Konfigurationsmanagement und Test
- Konfiguration von Produktlinien
- Literatur: Clements und Northrop (2001)

Empirisches Software-Engineering

- Empirische Forschung in der Softwaretechnik
- Methoden
- Literatur: Endres und Rombach (2003); Prechelt (2001); Yin (2003)

Allgemeine Literatur zur Softwaretechnik:

- Sommerville (2004)
- Pressman (1997)
- Balzert (1997)
- Ludewig und Lichter (2006)

Termine

- dienstags, 10:15 – 11:45 Uhr, MZH 5210
- mittwochs, 14:00 s.t. – 15:30 Uhr, MZH 1470

Übungen und Ressourcen

Dozent:

- Erreichbar: TAB 2.57, Telefon 218-64481, `koschke@tzi.de`
- `http://www.informatik.uni-bremen.de/~koschke/`
- Sprechstunde nach Vereinbarung

Ressourcen:

- annotierte Folien unter `http://www.informatik.uni-bremen.de/st/lehredetails.php?id=&lehre_id=412`
- in der Vorlesung gezeigte und mit Tablet-PC beschriftete Folien in Stud.IP → registrieren!
- Videoaufzeichnungen aus dem Jahr 2007 unter `http://mlecture.uni-bremen.de/`
- News und annotierte Folien unter Stud.IP unter `http://elearning.uni-bremen.de`

Übungen:

- Übungen ca. alle zwei Wochen alternierend zur Vorlesung
- Übungsblatt im Netz

Scheinbedingungen

Anerkennung durch mündliche Prüfung:

- 30 minütige mündliche Prüfung über den Stoff der Vorlesung
- Übungsaufgaben bearbeiten lohnt sich

Lehrbücher I

Allgemeine Literatur zur Softwaretechnik

- Sommerville (2004)
- Pressman (1997)
- Balzert (1997)
- Ludewig und Lichter (2006)

Software-Metriken

- Fenton und Pfleeger (1998)

Aufwand- und Kostenschätzung

- Boehm u. a. (2000)
- Poensgen und Bock (2005)

Lehrbücher II

Software-Entwicklungsprozesse

- Beck (2000)
- Kruchten (1998)
- Balzert (2008)
- Bunse und von Knethen (2002)
- Pichler (2008)
- auch: allgemeine Literatur über Softwaretechnik

Software-Prozessverbesserung

- Siviy u. a. (2007)
- Kneuper (2006)

Komponentenbasierte Entwicklung

- Szyperski u. a. (2002)

Lehrbücher III

Modellgetriebene Entwicklung

- Stahl u. a. (2007)

Software-Architektur

- Bass u. a. (2003)
- Hofmeister u. a. (2000)

Entwurfsmuster

- Gamma u. a. (2003)
- Buschmann u. a. (1996)
- Schmidt u. a. (2000)

Software-Produktlinien

- Clements und Northrop (2001)

Lehrbücher IV

Empirisches Software-Engineering

- Endres und Rombach (2003)
- Yin (2003)
- Prechelt (2001)

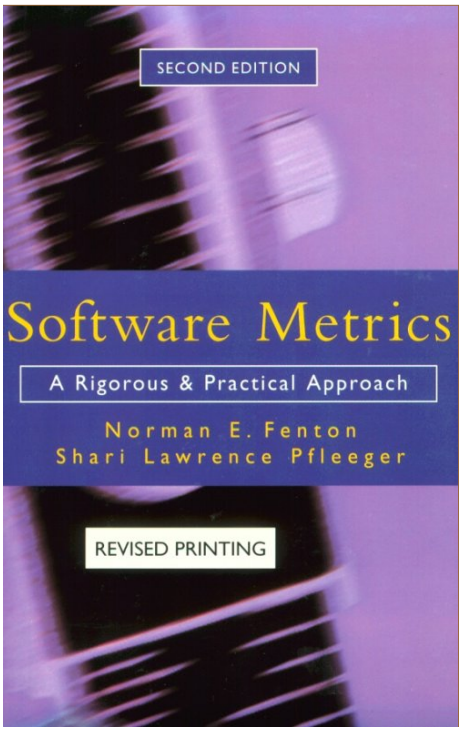
Software-Metriken

Software-Metriken
Messen und Maße
Skalen
Gütekriterien für Metriken
Vorgehensweise
Klassifikation von Softwaremetriken
Prozessmetriken
Ressourcenmetriken
Produktmetriken
Anwendungen
Probleme
Goal-Question-Metric-Ansatz
Wiederholungsfragen

Lernziele

- Verstehen, was eine Software-Metrik ist
- die Einsatzmöglichkeiten von Metriken kennen
- Metriken beurteilen und auswählen können

Literatur



– Fenton und Pfleeger (1998)

Bedeutung des Messens

"To measure is to know." Clerk Maxwell, 1831-1879

"A science is as mature as its measurement tools."
Louis Pasteur, 1822-1895

"Miss alles, was sich messen lässt, und mach alles messbar, was sich nicht messen lässt." Galileo Galilei, 1564-1642

"Messen können Sie vieles, aber das Angemessene erkennen können Sie nicht."

Messen spielt in allen Ingenieurswissenschaften eine wichtige Rolle.
Galilei: Ziel der Wissenschaft; durch Messung zu verständlicheren und nachprüfbaren Konzepten/Ergebnissen kommen.

Definitionen: Maß, Messen, Metrik

Definition
Maß:
Abbildung von einem beobachteten (empirischen) Beziehungssystem auf ein numerisches Beziehungssystem

Abbildung von Eigenschaften von Objekten der realen Welt auf Zahlen oder Symbole

Definition
Messen: Anwendung eines Maßes auf ein Objekt

Definition
Metrik: Abstandsmaß (math.)

Definitionen für Software-Metriken

> *"A quantitative measure of the degree to which a system, component, or process possesses a given variable."*
>
> – IEEE Standard Glossary

> *"A software metric is any type of measurement which relates to a software system, process or related documentation."*
>
> – Ian Summerville

Messen und Softwaretechnik

Zwecke:

- Beschreibung: kompakt und objektiv
- Beurteilung: Vergleich, Verbesserungen
- Vorhersage: geordnete Planung, Verbesserungen

Fragen bei Maßen

• Repräsentanz	Darstellung als Zahl sinnvoll möglich?
• Eindeutigkeit	viele Abbildungen möglich
• Bedeutung	erhalten bei Transformationen
• Skalierung	welche Skala?

Worüber wir uns bei der Definition von Metriken Gedanken machen müssen.

There are three important questions concerning representations and scales:

1. How do we determine when one numerical relation system is preferable to another?
2. How do we know if a particular empirical relation system has a representation in a given numerical relation system?
3. What do we do when we have several different possible representations (and hence many scales) in the same numerical relation system?

Question 2 is known as the representation problem.

— Fenton und Pfleeger (1998)

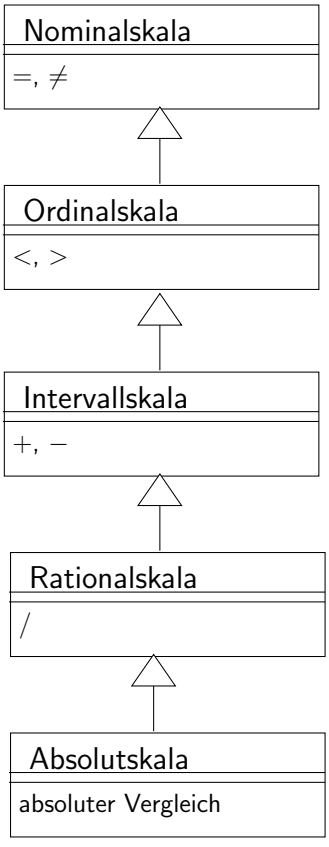
Skalen

1. „20 Prozent Verbesserung der Qualität“
2. „Dieser Kunde ist doppelt so zufrieden wie jener“
3. „Heute ist es doppelt so warm wie gestern“
   (Temperatur gestern: 10°C; heute: 20°C)

1. Was ist Qualität Null?
2. Wie zufrieden sind sie denn?
3. $10°C \rightarrow 20°C \mathrel{\hat{=}} +3{,}5\%$
   denn 10°C = 283 Kelvin, 20°C = 293 Kelvin

$\rightarrow$ Skala?

Skalenhierarchie

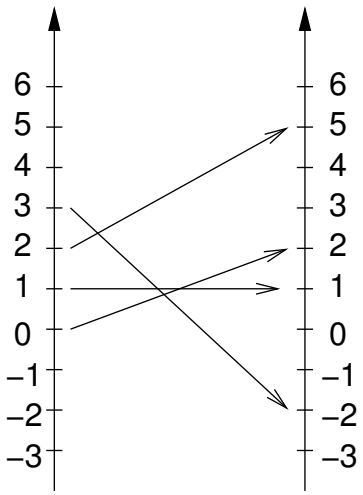


Skalenhierarchie – Nominalskala

1. Nominalskala

- ungeordnete 1:1 Abbildung
- Operationen: $=$, $\neq$
- Statistiken: Häufigkeit
- Transformationen: beliebige 1:1
- Beispiel: Programmiersprachen

  Ada C C++ Java ...

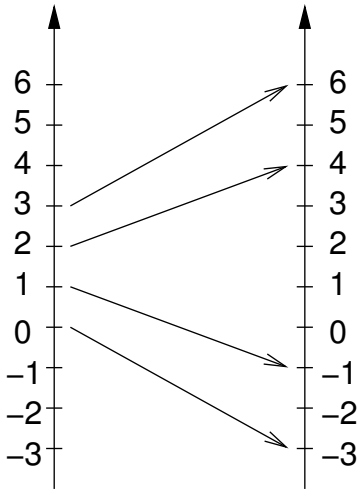


Skalenhierarchie – Ordinalskala

2. Ordinalskala

- dazu: vollständige Ordnung
- Transformationen: streng monoton steigend
- Operationen: $<$, $>$
- Statistiken: Median
- Beispiel: Prioritäten

  niedrig $<$ mittel $<$ hoch

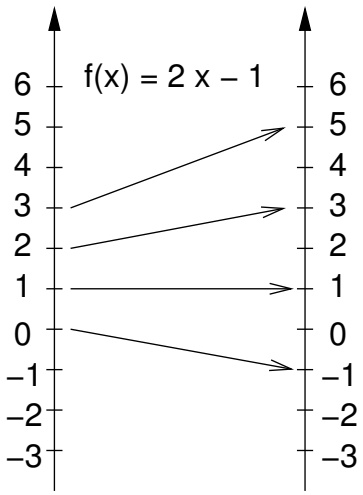


Skalenhierarchie – Intervallskala

3. Intervallskala

- dazu: Distanzfunktion
- Transformationen: $M' = aM + b \quad (a > 0)$
- Operationen: $+$, $-$
- Statistiken: Mittelwert, Standardabweichung
- Beispiel: Temperatur

$$T_{Celsius} = \frac{5}{9} \cdot (T_{Fahrenheit} - 32)$$



Definition Metrik

Metrik: Distanzfunktion $d : A \times A \to IR$, mit:

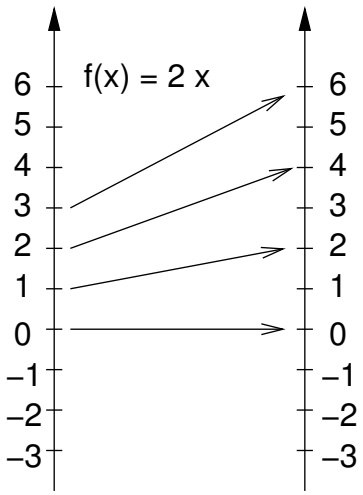
- $d(a, b) \geq 0 \ \forall a, b \in A, \quad d(a, b) = 0 \Leftrightarrow a = b$
- $d(a, b) = d(b, a) \ \forall a, b \in A$
- $d(a, c) \leq d(a, b) + d(b, c) \ \forall a, b, c \in A$

Skalenhierarchie – Rationalskala

4. Rationalskala

- dazu: Maßeinheit, Nullpunkt
- Transformationen: $M' = aM \quad (a > 0)$
- Operationen: $/$
- Statistiken: geom. Mittel, Korrelation
- Beispiel: Länge

$$L_{Meter} = L_{Meilen} \cdot 1609$$

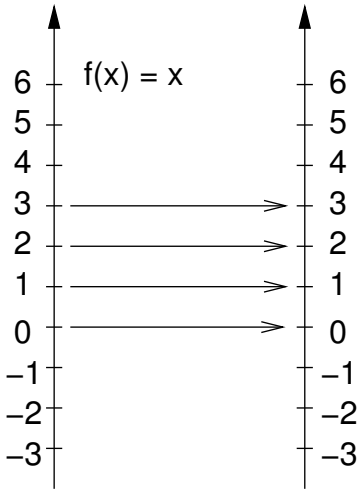


Das geometrische Mittel zwischen zwei Zahlenwerten ist: $\sqrt{f1 \cdot f2}$
Das arithmetische Mittel zwischen zwei Zahlwerten ist: $(f1 + f2)/2$

Skalenhierarchie – Absolutskala

5. Absolutskala

- Metrik steht für sich selbst, kann nicht anders ausgedrückt werden
- Transformationen: nur die Identität $M' = M$
- Operationen: absoluter Vergleich; d.h
  - es existiert ein natürlicher Nullpunkt
  - und Maßeinheit ist natürlich gegeben (d.h. im weitesten Sinne 'Stück')
- Beispiele:
  - Zähler: Anzahl Personen in einem Projekt
  - Wahrscheinlichkeit eines Fehlers
  - LOC für Anzahl Codezeilen
  - **nicht:** LOC für Programmlänge



Gütekriterien für Metriken

- Objektivität
- Validität
- Zuverlässigkeit
- Nützlichkeit
- Normiertheit
- Vergleichbarkeit
- Ökonomie

– Balzert (1997)

(Güte entspr. Qualität)
Objekt.: kein subjektiver Einfluss durch Messenden möglich
Valid.: misst wirklich das, was sie vorgibt zu messen
Zuverl.: Wiederholung liefert gleiche Ergebnisse
Nützl.: hat praktische Bedeutung
Norm.: es gibt eine Skala für die Messergebnisse
Vergl.: mit anderen Maßen vergleichbar; man kann das Maß mit anderen Maßen in eine Relation setzen
Ökon.: mit vertretbaren Kosten messbar

Vorgehensweise

1. Definition
   - Zielbestimmung
   - Modellbildung
   - Skalentypbestimmung
   - Maßdefinition
2. Validierung
3. Anwendung
   - Konkretes Modell bilden
   - Messung
   - Interpretation
   - Schlussfolgerung

Validierung von Maßen

Interne Validierung:

Nachweis, dass ein Maß eine gültige numerische Charakterisierung des entsprechenden Attributs ist, durch

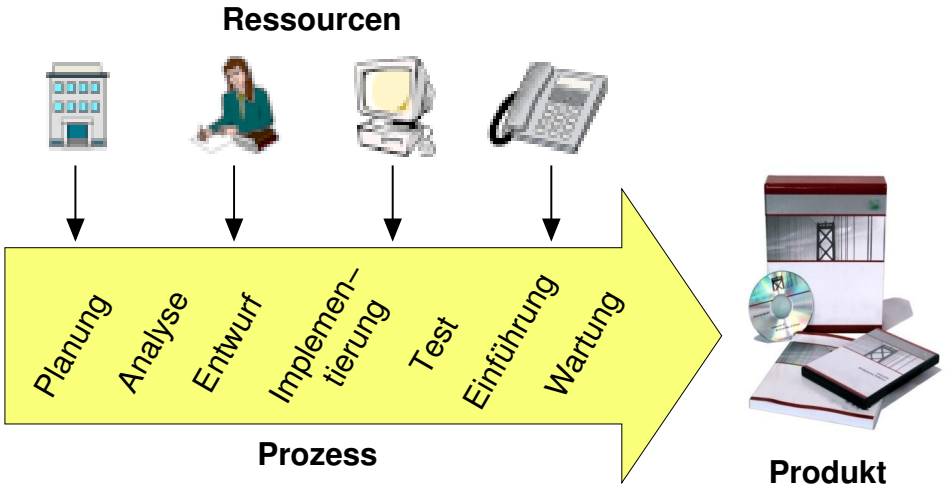
- Nachweis der Erfüllung der Repräsentanzbedingung
- und Prüfung des Skalentyps

Externe Validierung → Vorhersagemodell:

- Hypothese über Zusammenhang zwischen zwei Maßen
- Erfassung der Meßwerte beider Maße auf gleicher Testmenge
- Statistische Analyse der Ergebnisse
  → Bestimmung von Parametern
  → Prüfung der Allgemeingültigkeit

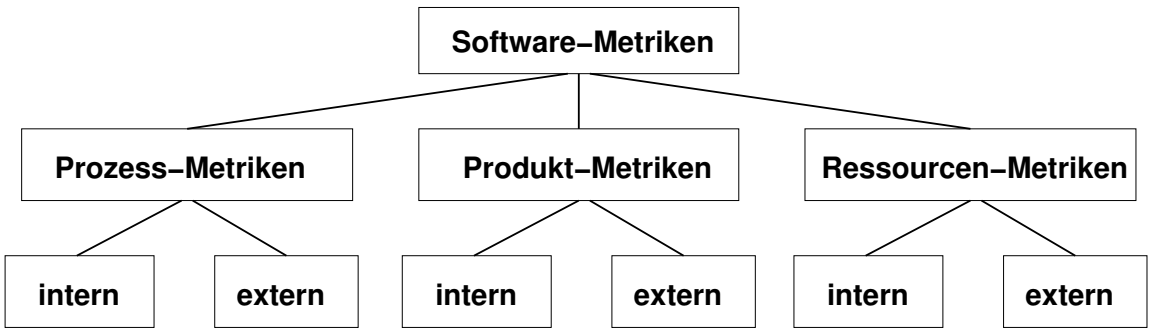
Klassifikation von Softwaremetriken

- Was: Ressource/Prozess/Produkt
- Wo: intern/extern (isoliert/mit Umgebung)
- Wann: in welcher Phase des Prozesses
- Wie: objektiv/subjektiv, direkt/abgeleitet

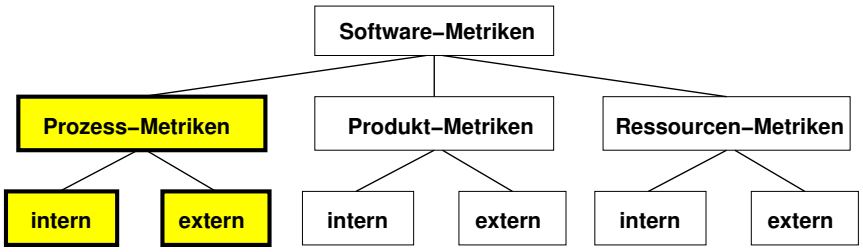


Bei den Metriken unterscheidet man zwischen internen und externen Metriken. Eine interne Metrik ist darüber definiert, dass sie nur Eigenschaften innerhalb des untersuchten Objektes misst, wohingegen externe Metriken die Interaktion des Objektes mit seiner Umgebung berücksichtigen.

Klassifikation nach Fenton und Pfleeger (1998)



Prozessmetriken



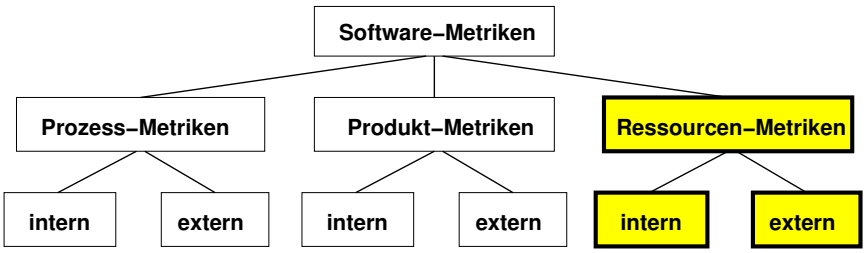
intern:

- Zeit/Dauer
- Aufwand
- Anzahl von Ereignissen
  z.B. Fehler, Änderungen

extern:

- Qualität
- Kontrollierbarkeit
- Stabilität
- Kosten

Ressourcenmetriken



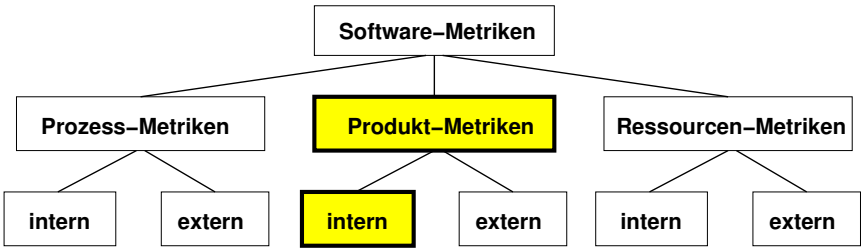
intern:

- Personal (Alter, Lohn)
- Teamgröße/-struktur
- Produktionsmaterialien
- Werkzeuge, Methoden

extern:

- Produktivität
- Erfahrung
- Kommunikation
- ...

Produktmetriken – intern



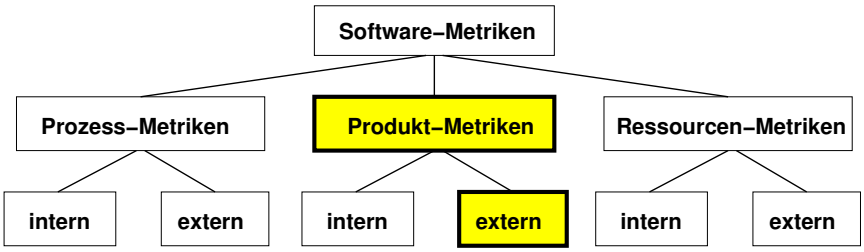
Größe:

- LOC
- Halstead
- Function Points
- Bang (DeMarco)

Komplexität:

- McCabe Cyclomatic Complexity
- Kontrollflussgraph
- Datenfluss
- OO-Metriken

Produktmetriken – extern



- Zuverlässigkeit
- Verständlichkeit
- Benutzerfreundlichkeit
- Performanz
- Portierbarkeit
- Wartbarkeit
- Testbarkeit
- ...

Produktmetriken – intern

Vorteil: automatische Erfassung

Die Klassiker:

- LOC - Lines Of Code
- Halstead (1977)
- McCabe (1976)
- OO-Metriken (Chidamber und Kemerer 1994)

Größenmetriken – LOC

Lines of code (LOC)

+ relativ einfach messbar
+ starke Korrelation mit anderen Maßen
– ignoriert Komplexität von Anweisungen und Strukturen
– schlecht vergleichbar
- abgeleitet: Kommentaranteil

Physical source lines (COCOMO 2.0)

When a line or statement contains more than one type, classify it as the type with the highest precedence.

Statement type	Precedence	Included
Executable	1	√
Nonexecutable		
Declarations	2	√
Compiler directives	3	
Comments		
On their own lines	4	
On lines with source code	5	
Banners and non-blank spacers	6	
Blank (empty) comments	7	
Blank lines	8	

Physical source lines (COCOMO 2.0)

How produced	Included
Programmed	√
Generated with source code generators	
Converted with automated translators	√
Copied or reused without change	√
Modified	√
Removed	

Physical source lines (COCOMO 2.0)

Origin	Included
New work: no prior existence	√
Prior work: taken or adapted from	√
A previous version, build, or release	√
Commercial off-the-shelf software (COTS), other than libraries	
Government furnished software (GFS), other than reuse libraries	
Another product	
A vendor-supplied language support library (unmodified)	
A vendor-supplied operating system or utility (unmodified)	
A local or modified language support library or operating system	
Other commercial library	
A reuse library (software designed for reuse)	√
Other software component or library	√

Anwendungen

- Beurteilung des aktuellen Zustands
  - Projektüberwachung
  - Produktivität
  - Softwarequalität
  - Prozessqualität (CMM)
- Vorhersage des zukünftigen Zustands
  - Aufwandsabschätzung
  - Prognose für Wartungskosten

Probleme

- Datenerfassung sehr aufwendig, zunächst wenig Nutzen
- Datenerfassung nicht konsistent
- Teilweise Messungen schwierig durchführbar
- Zweck der Messungen muss klar sein
- Integration der Datenerfassung in den normalen Arbeitsprozess
- Metriken müssen wohldefiniert und validiert sein
- Beziehungen zwischen Metriken müssen definiert sein
- Gefahr der Fehlinterpretation

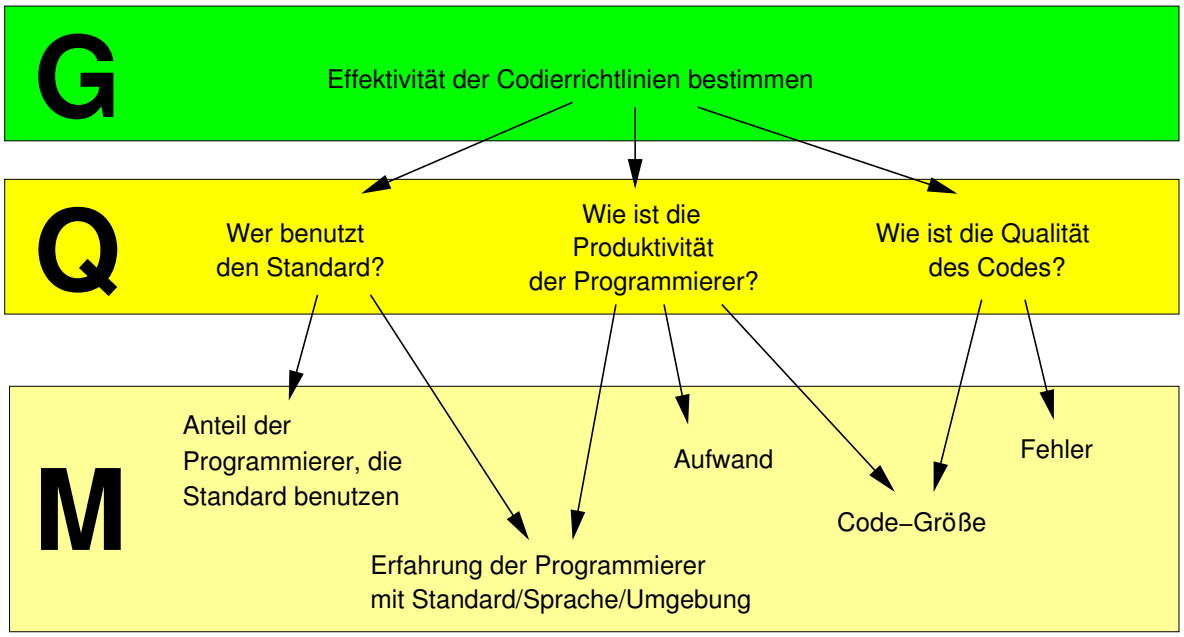
Zielorientiertes Messen

Goal-Question-Metric (GQM) (Basili und Weiss 1984))

1. Ziele erfassen
2. zur Prüfung der Zielerreichung notwendige Fragen ableiten
3. was muss gemessen werden, um diese Fragen zu beantworten

Nicht das messen, was einfach zu bekommen ist, sondern das, was benötigt wird.

Zielorientiertes Messen



Beispiel: Prozess

Ziel	Frage	Metrik
Maximiere Kundenzufriedenheit	Wie viele Probleme treten beim Kunden auf?	• #Fehler (FR) und #Änderungswünsche (ÄR) • Zuverlässigkeit • Break/Fix-Verhältnis
	Wie lange dauert Problembehebung?	• Verhältnis und Dauer offener und geschlossener FR/ÄR
	Wo sind Flaschenhälse?	• Personalnutzung • Nutzung anderer Ressourcen

Wiederholungs- und Vertiefungsfragen

- Was ist ein Maß? Was ist eine Metrik?
- Was ist eine Software-Metrik?
- Welche Skalen gibt es für Daten? Welche Eigenschaften haben diese?
- Beschreiben Sie das Vorgehen bei der Definition und Einführung eines Maßes. Was unterscheidet die interne von der externen Validierung?
- Wie lassen sich Software-Metriken klassifizieren? Nennen Sie Beispiele für jede Klasse.
- Was ist die Bedeutung von Metriken im Software-Entwicklungsprozess?
- Was ist die GQM-Methode? Erläutern Sie GQM anhand des Zieles X.

N.B.: Die Übungsaufgaben sind weitere Beispiele relevanter Fragen.

Entwicklungsprozesse

Entwicklungsprozesse
- Lernziele
- Wasserfallmodell
- V-Modell
- Testgetriebene Entwicklung
- Inkrementelle Entwicklung
- Spiralmodell
- Rational Unified Process
- Cleanroom Development
- Agile Methoden
- Extreme Programming (XP)
- Scrum
- Agile versus traditionelle Vorgehensweisen
- Capability Maturity Model
- Persönlicher Softwareprozess
- Wiederholungsfragen
- Weiterführende Literatur

Lernziele

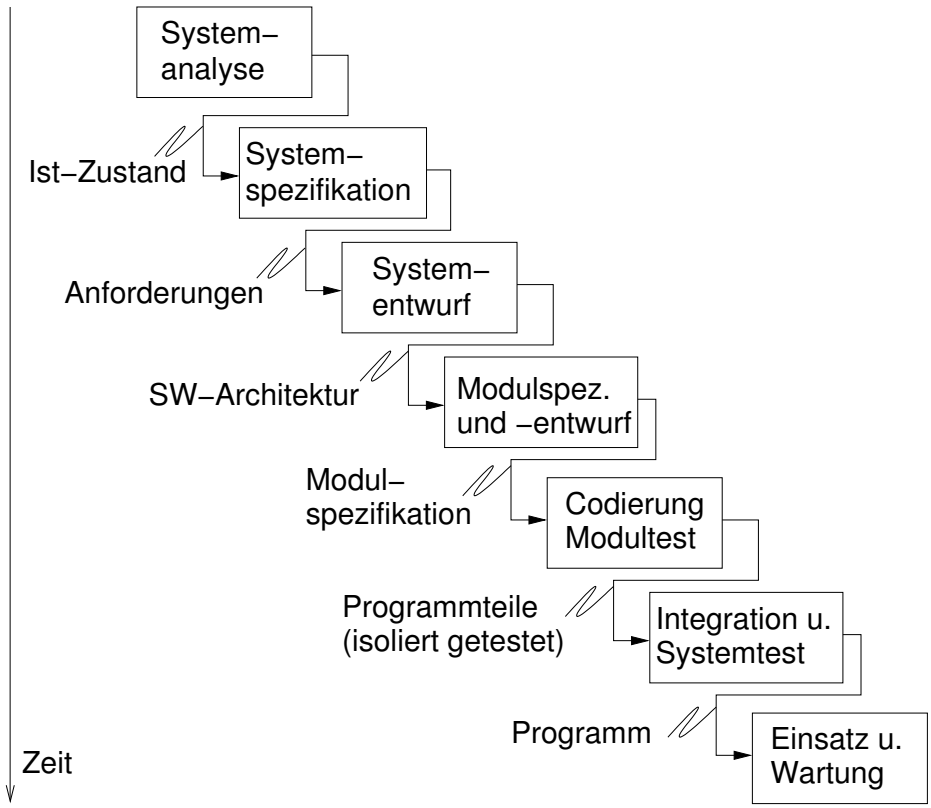
- verschiedene Software-Entwicklungsprozessmodelle kennen
- Vor- und Nachteile/Anwendbarkeit abwägen können
- die Besonderheit von Prozessen der Software-Entwicklung kennen

Entwicklungsprozesse

Grundlagen für guten Prozess:

- Wohldefiniertheit
  - sonst Falsch-, Nicht-, Mehrarbeit
  - sonst Information nicht verfügbar oder unverständlich
- Quantitatives Verstehen
  - objektive Grundlage für Verbesserungen
- Kontinuierliche Änderung
  - sonst Prozess schnell überholt
- Angemessenheit
  - Prozess muss dem zu lösenden Problem angemessen sein
  - d.h. effektiv und effizient sein

Striktes Wasserfallmodell nach Royce (1970)



Hier sehen wir als Beispiel das strikte Wasserfallmodell. Es enthält alle Aktivitäten in streng sequenzieller Folge. Dieses Modell eignet sich, wenn die Arbeitsschritte deutlich voneinander getrennt werden können. Alle Risiken müssen vor Projektbeginn ausgeschlossen werden können. Es ist geeignet, wenn die Mitarbeiteranzahl klein ist, da alle Mitarbeiter gleichzeitig an einem Arbeitsschritt arbeiten können.
Dieses Modell ist für die allermeisten Aufgabenstellung jedoch unrealistisch. Es setzt voraus, dass jede Aktivität auf Anhieb erfolgreich abgeschlossen werden kann. Allerdings werden z.B. die Anforderungen oft auch noch in späten Phasen des Projekts geändert bzw. erst dann überhaupt erst richtig verstanden. Dann müssen frühe Aktivitäten wiederholt werden.

Striktes Wasserfallmodell Royce (1970)

Eigenschaften dieses Modells:

- dokumenten-getrieben: jede Aktivität erzeugt Dokument
- streng sequenzielle Aktivitäten

\+ klar organisierter Ablauf

\+ relativ einfache Führung

\+ hohe Qualität der Dokumentation

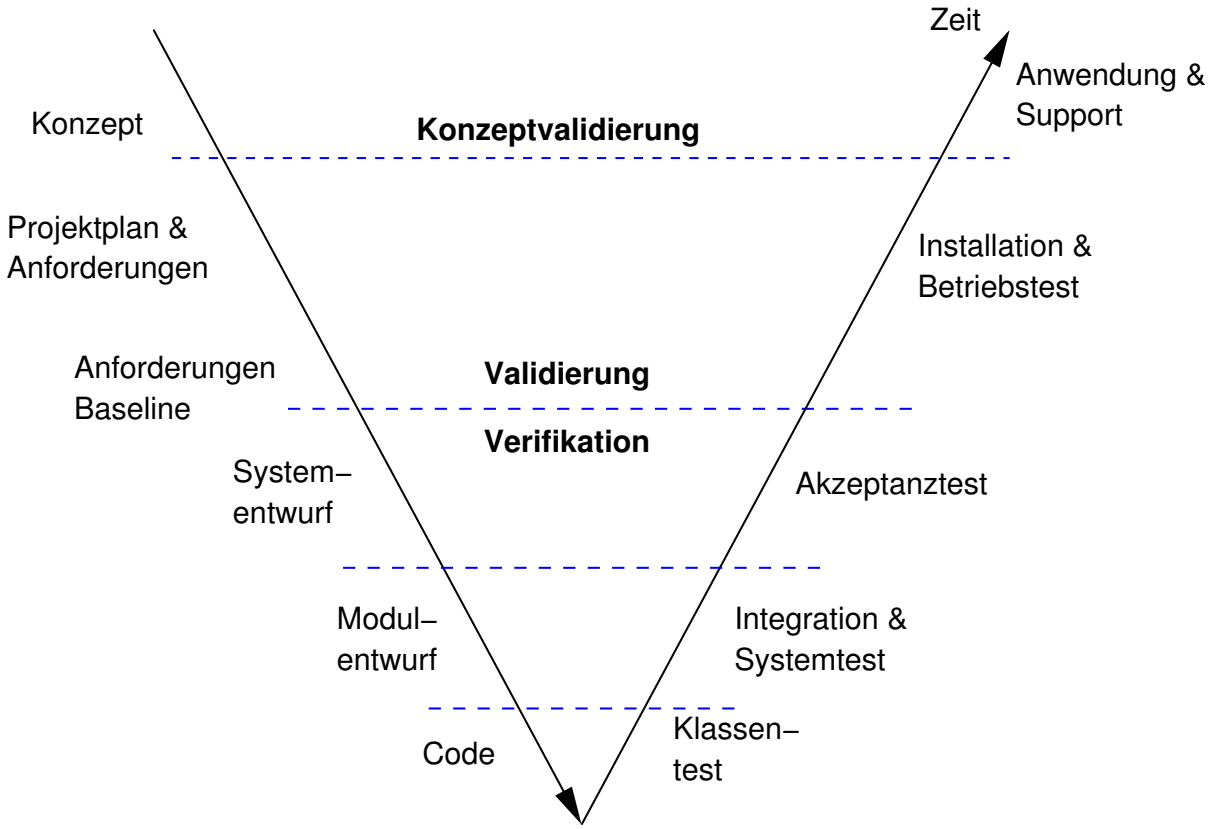
– Entwicklung bildet langen Tunnel

– 90%-fertig-Syndrom

– Spezifikationsmängel lassen sich kaum frühzeitig erkennen

– Entwicklung beim Kunden wird ignoriert

V-Modell von Boehm (1979)



Das V-Modell ist kein eigenständiges Prozessmodell im eigentlichen Sinn. Die Produkte im V-Modell werden wie im Wasserfallmodell streng sequenziell erstellt. Es fügt dem Wasserfallmodell lediglich eine zusätzliche strukturelle Sicht hinzu, nämlich dass für jedes zu erstellende Dokument ein entsprechender Test existieren und durchgeführt werden soll.
Das V-Modell sieht sowohl Verifikationen als auch Validierungen vor. Validierung: Es wird das richtige Produkt erstellt. Verifikation: Das Produkt wird richtig erstellt.
Eine weitere Konsequenz des V-Modells für die konkrete Projektplanung ist, dass man die Test- und Validierungsaktivitäten früher als die tatsächliche Durchführung der Aktivität vorbereiten kann. Beispielsweise kann der Akzeptanztest vorbereitet werden, sobald man die Anforderungen kennt. Auf diese Weise können Aktivitäten bei einem Projektteam parallelisiert werden: Der Tester kann Testfälle für den Akzeptanztest entwickeln, auch wenn noch keine Implementierung existiert.

V-Modell von Boehm (1979)

Eigenschaften:

- entspricht Wasserfallmodell
- \+ betont Qualitätssicherung
- \+ frühe Vorbereitung von Validierung und Verifikation
  - zusätzliche Parallelisierung
  - Fehler/Mängel werden früher entdeckt
- – alle sonstigen Nachteile des Wasserfallmodells

Testgetriebene Entwicklung (Sneed 2004)

Kennzeichen testgetriebener Entwicklung:

Testfälle . . .

- werden früh aus Anwendungsfällen abgeleitet
- dienen als Baseline
- treiben den Entwurf
- treiben die Kodierung

Test-Teams treiben die Entwicklung, statt von Entwicklern getrieben zu werden

Testgetriebene Entwicklung (Sneed 2004)

Anwendungs- und Testfälle

- Anwendungsfälle beschreiben Anforderungen
- sind jedoch oft nicht detailliert genug, um erwartetes Verhalten vollständig zu spezifizieren
- Testfälle können Anwendungsfälle hier komplementieren
- zu jedem Anwendungsfall sollte es mindestens einen Testfall geben
- Testfälle können zur Kommunikation zwischen Entwickler und Kunden/Anwender dienen

Testgetriebene Entwicklung (Sneed 2004)

Testfälle dienen als Baseline:

- definieren die Vorbedingungen einer Produktfunktion
- spezifizieren die Argumente einer Produktfunktion
- spezifizieren das Verhalten einer Produktfunktion (die Nachbedingungen)

Testgetriebene Entwicklung (Sneed 2004)

Testfälle treiben den Entwurf:

- Testfall ist ein Pfad durch die Software-Architektur
- Testfälle verknüpfen Anforderungsspezifikation und Architekturkomponenten
- Testfälle können zur Studie der zu erwartenden Performanz und anderer Systemattribute zur Entwurfszeit verwendet werden

Testgetriebene Entwicklung (Sneed 2004)

Testfälle treiben die Kodierung:

- vor Implementierung einer Klasse werden erst Testtreiber entwickelt
- Testtreiber enthält mindestens einen Test für jede nichttriviale Methode
- Testfall hilft, die richtige Schnittstelle zu definieren
- Testfall zwingt Entwickler, über das zu erwartende Resultat nachzudenken
- Testfälle spezifizieren die Methoden zumindest partiell

Testgetriebene Entwicklung (Sneed 2004)

Tester treiben die Entwicklung:

- Tester sind verantwortlich für die Auslieferung des Produkts
- Tester legen Kriterien für die Auslieferbarkeit fest
- Entwickler sind Lieferanten für die Tester
- Softwareentwicklung ist eine Versorgungskette; das Ende zieht, anstatt gedrückt zu werden

Bewusstseinsebenen

- bewusstes Wissen (20-30%)
  - Wissen, über das man sich im Klaren ist oder das in seiner vollen Bedeutung klar erkannt wird
- unbewusstes Wissen ($\leq$40%)
  - Wissen, das sich dem Bewusstsein im Moment nicht darbietet, aber dennoch handlungsbestimmend ist, und potenziell aufgerufen werden kann
- unterbewusstes Wissen
  - unbekannte Wünsche, die erst von außen herangetragen werden müssen, um als Anforderungen erkannt zu werden

- bewusst: will mit meinem Handy telefonieren
- unbewusst: Tastatur und Display sollen auf derselben Seite meines Handys sein
- unterbewusst: ich will SMS verschicken können

Basisfaktoren

- Minimalanforderungen

→ Mangel führt zu massiver Unzufriedenheit

→ mehr als Zufriedenheit ist nicht möglich

Leistungsfaktoren

- bewusst verlangte Sonderausstattung

→ bei Erfüllung: Kundenzufriedenheit

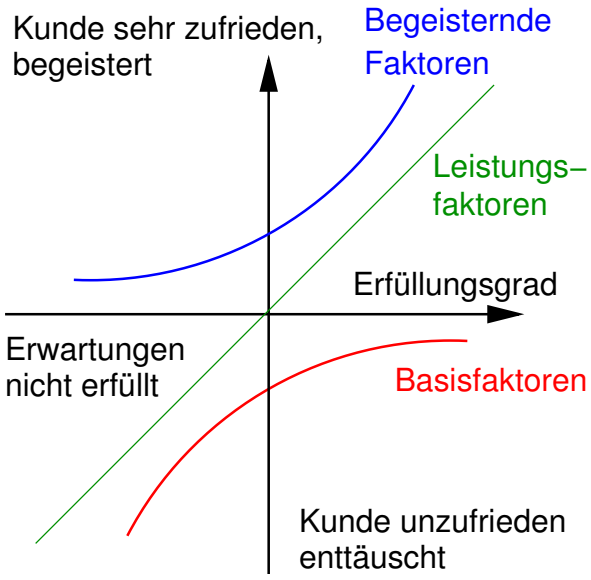
→ sonst: Unzufriedenheit

Begeisternde Faktoren

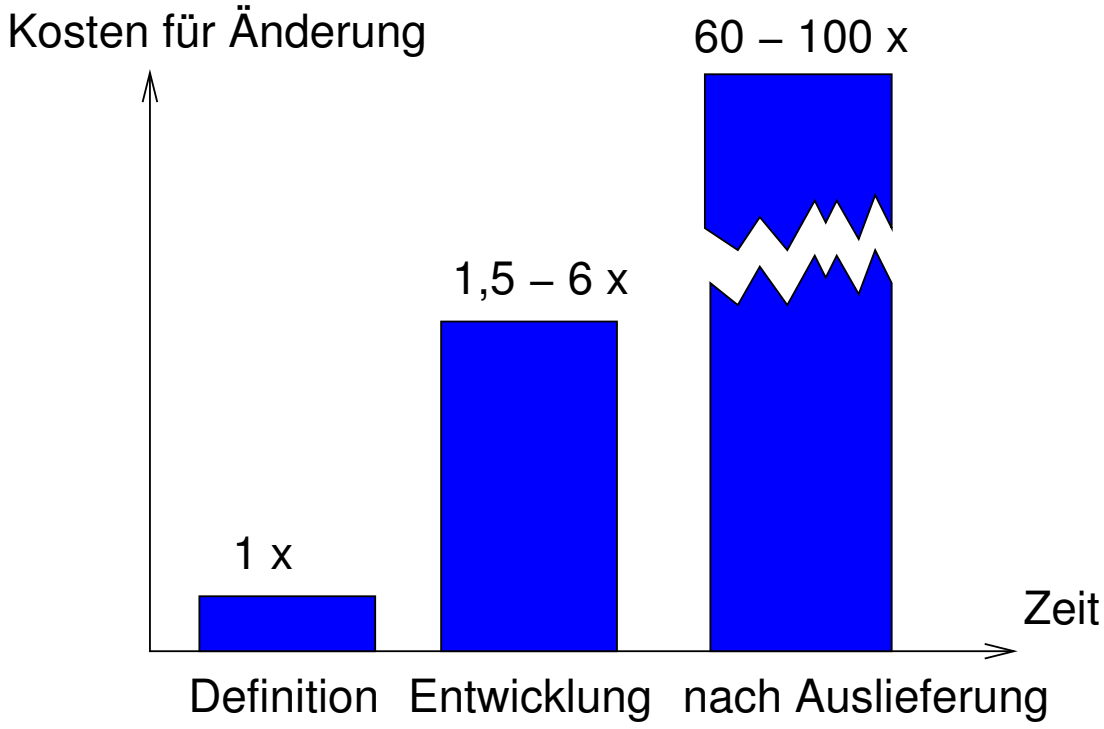
- unbewusste Wünsche, nützliche/angenehme Überraschungen

→ steigern Zufriedenheit überproportional

Kano-Modell

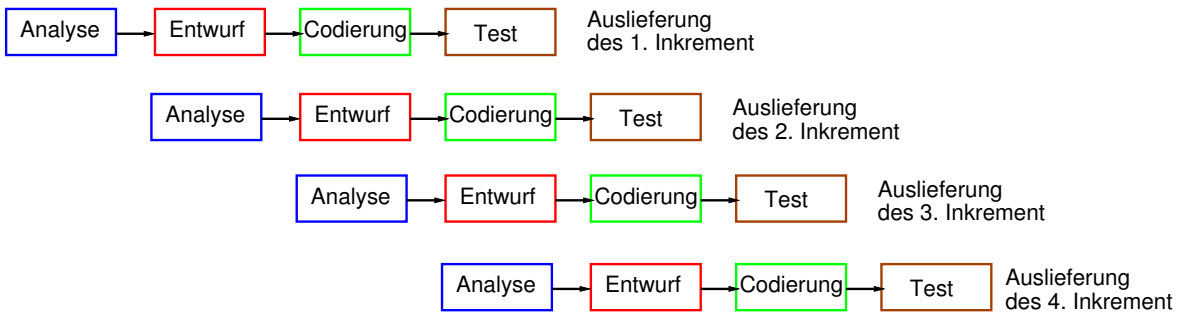


Kosten für Änderungen



Pressman (1997)

Inkrementelles Modell von Basili und Turner (1975)



Das Wasserfallmodell geht davon aus, dass alle Anforderungen zu Beginn erfasst werden können und diese während des Projekts stabil bleiben. Dies ist in der Praxis selten der Fall.
Im Laufe des Projekts gewinnen die Entwickler ein besseres Verständnis der Anwendungsdomäne und entdecken Irrtümer in ihren ursprünglichen – oft nur impliziten – Annahmen. Ähnlich wird auch der Kunde sich oft erst im Laufe des Projekts im Klaren, was er eigentlich erwarten kann und will. Auch die Rahmenbedingungen, unter denen das Projekt startete, können sich ändern (z.B. Gesetze oder Normen).
Die Anforderungen sind also selten stabil. Damit wird das Wasserfallmodell in seiner strikten Auslegung fragwürdig. Allerdings hat auch schon der Erfinder des Wasserfallmodells, Royce, empfohlen, das System zweimal zu entwickeln. Das erste Mal, um überhaupt erst Anforderungen und mögliche Lösungen auszuloten. Das zweite Mal, um ein adäquates und qualitativ hochwertiges Produkt zu erstellen.
Das inkrementelle Modell führt diesen Gedanken fort. Anstatt auf die Fertigstellung der gesamten Anforderungen zu warten, werden – sobald eine ausreichende Anzahl von Kernanforderungen beschrieben ist – für diese ein Entwurf und die Implementierung gestattet. Je mehr Anforderungen hinzukommen, desto mehr zusätzliche Inkremente werden gestartet.
Das inkrementelle Modell ist gut geeignet, wenn Kunde die Anforderungen noch nicht vollständig überblickt bzw. sich der Möglichkeiten zur Realisierung der Anforderungen nicht bewusst ist und deshalb die Anforderungen nicht formulieren kann.
Es ist mit diesem Modell möglich, Teile des Systems bereits vor Fertigstellung des gesamten Systems beim Kunden einzuführen (z.B. ein oder mehrere Subsysteme). Mit diesen Teilen kann der Kunde eine eingeschränkte Anzahl an Anforderungen realisieren. Die Zeitspanne zwischen Auftragsvergabe und Einsatz von zumindest Systemteilen wird somit geringer.
Federt die Gefahr des Wasserfallmodells ab, am Ende mit leeren Händen dazustehen. Sind wenigstens die ersten Inkremente erfolgreich entstanden, hat der Kunde zumindest ein partielles System.
Während der Entwicklung kann noch auf Änderungen reagiert werden.
Das Modell steht und fällt mit der Möglichkeit, Kernanforderungen zu identifizieren und ausreichend zu spezifizieren. Die initale Software-Architektur muss für alle Anforderungen – Kernanforderungen wie alle anderen, die im Laufe des Projekts formuliert werden – tragfähig sein; ansonsten ist ein hoher Restrukturierungaufwand notwendig. Darum sollten am Anfang alle Anforderungen bekannt sein, die sich maßgebend auf die Architektur auswirken.

Beispiel für Textverarbeitung

Iterationen:

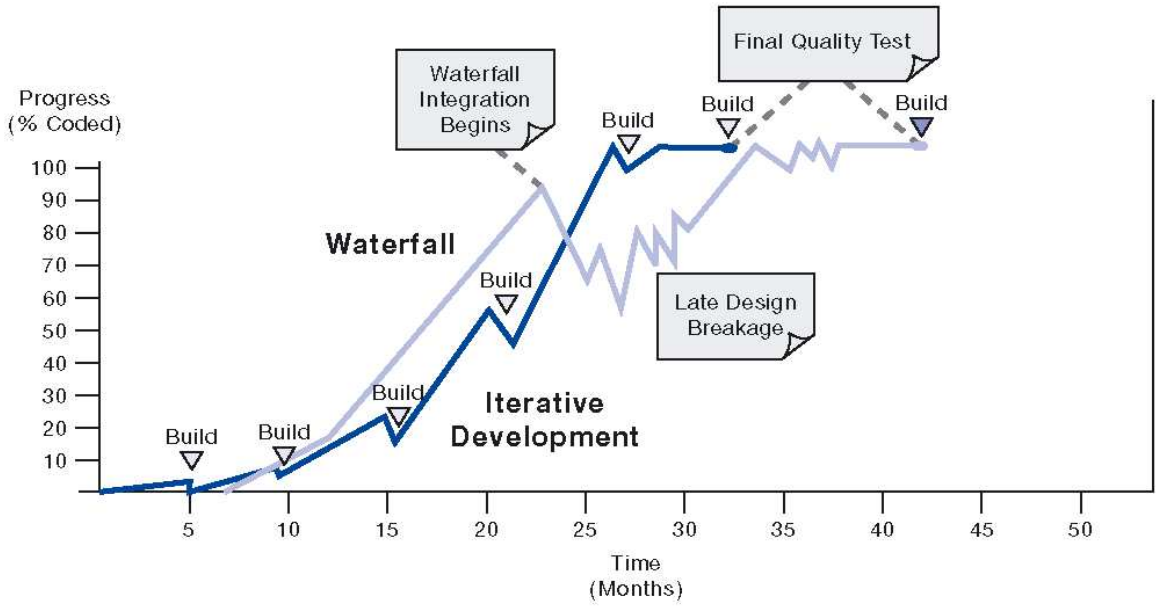
1. grundlegende Funktionalität
   - Datei-Management, Editor, Textausgabe
2. erweiterte Funktionalität
   - Style-Files, Bearbeitung mathematischer Formeln, Einbinden von Graphiken
3. zusätzliche Funktionalität
   - Rechtschreibprüfung, Grammatiküberprüfung, Überarbeitungsmodus
4. ergänzende Funktionalität
   - Tabellenkalkulation, Geschäftsgraphiken, E-Mail, Web-Browser, Scanner-Anbindung, Flipper

Inkrementelles Modell von Basili und Turner (1975)

Eigenschaften:

- Wartung wird als Erstellung einer neuen Version des bestehenden Produkts betrachtet.
- \+ Entwicklung erfolgt stufenweise
  - brauchbare Teillösungen in kurzen Abständen
- \+ Lernen durch Entwicklung und Verwendung des Systems.
- \+ Gut geeignet, wenn Kunde Anforderungen noch nicht vollständig überblickt oder formulieren kann.
  - *"I know it when I see it"*
- – Kernanforderungen und Architekturvision müssen vorhanden sein.
- – Entwicklung ist durch existierenden Code eingeschränkt.

Vergleich inkrementelles Modell und Wasserfallmodell



Spiralmodell von Boehm (1988)

Mehrere Iterationen der folgenden Schritte:

1. **Bestimmung der Ziele** und Produkte des Durchlaufs; Berücksichtigung von Alternativen (z.B. Entwurfsvarianten) und Restriktionen (z.B. Zeitplan)
2. **Bewertung der Risiken** für alle Alternativen; Entwicklung von Lösungsstrategien zur Beseitigung der Ursachen
3. **Arbeitsschritte durchführen**, um Produkt zu erstellen
4. Review der Ergebnisse und **Planung** der nächsten Iteration

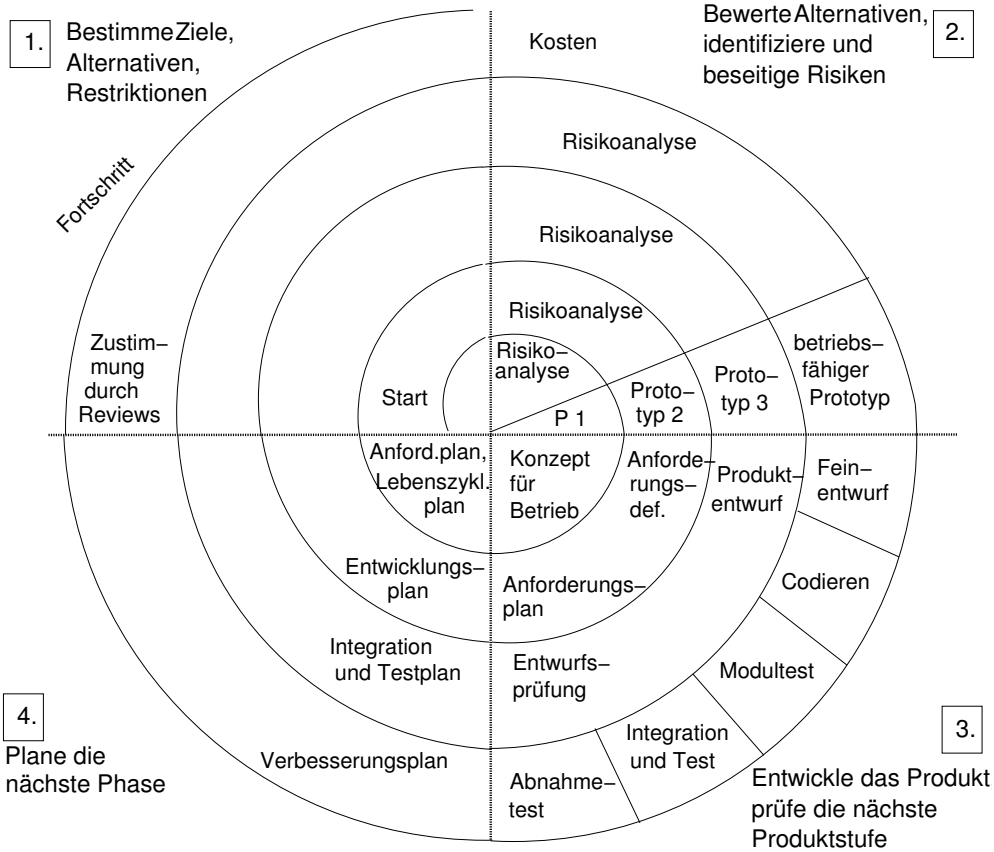
Das Spiralmodell kann für sehr große und komplexe Projekte verwendet werden, da es der Komplexität durch das risikogesteuerte Vorgehen Rechnung trägt. Die Anzahl der Durchläufe ergibt sich erst während des Projekts und wird durch die auftetenden Risiken bestimmt. Dies hat zur Folge, dass zu Beginn des Projekts ein Zeit- und Kostenplan nur schwer zu erstellen ist. Die Risikoanalyse kann nur durch erfahrene Projektleiter durchgeführt werden. Bei zu zaghaftem Vorgehen kann sich das Projekt unnötigerweise verlängern, was zu erhöhten Kosten führt. Zu schnelles Vorgehen kann Risiken vernachlässigen und zu Problemen in folgenden Durchläufen führen.

Beispiel eines Spiralmodells

(Generische) Risiken:

- Ist das Konzept schlüssig? Kann es aufgehen?
- Was sind die genauen Anforderungen?
- Wie sieht ein geeigneter Entwurf aus?

Beispiel eines Spiralmodells mit vier Durchläufen



Bewertung Spiralmodell

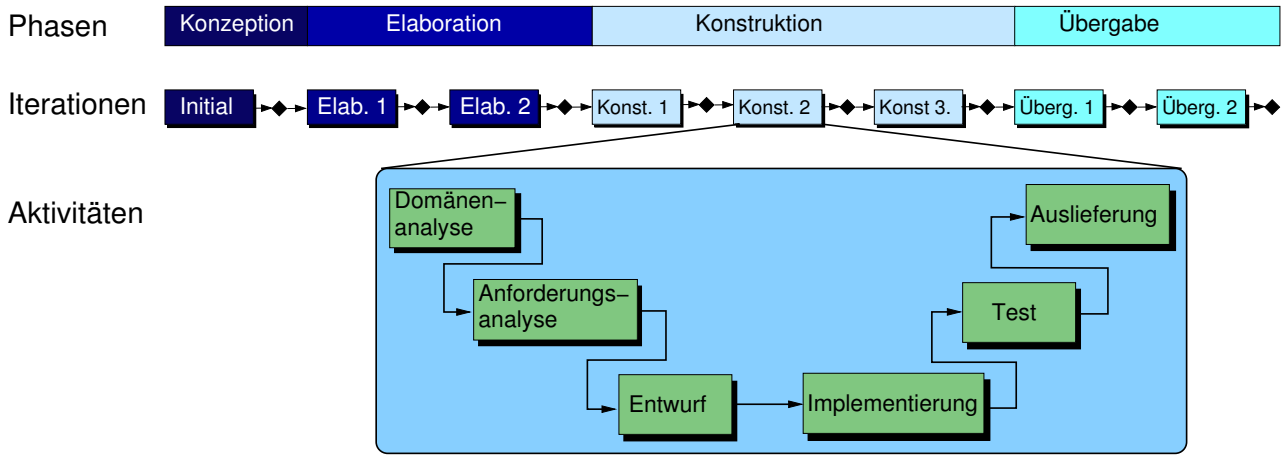
- Meta-Modell: Iterationen können beliebigen Modellen folgen

\+ bei unübersichtlichen Ausgangslagen wird die Entwicklung in einzelne Schritte zerlegt, die jeweils unter den gegebenen Bedingungen das optimale Teilziel verfolgen

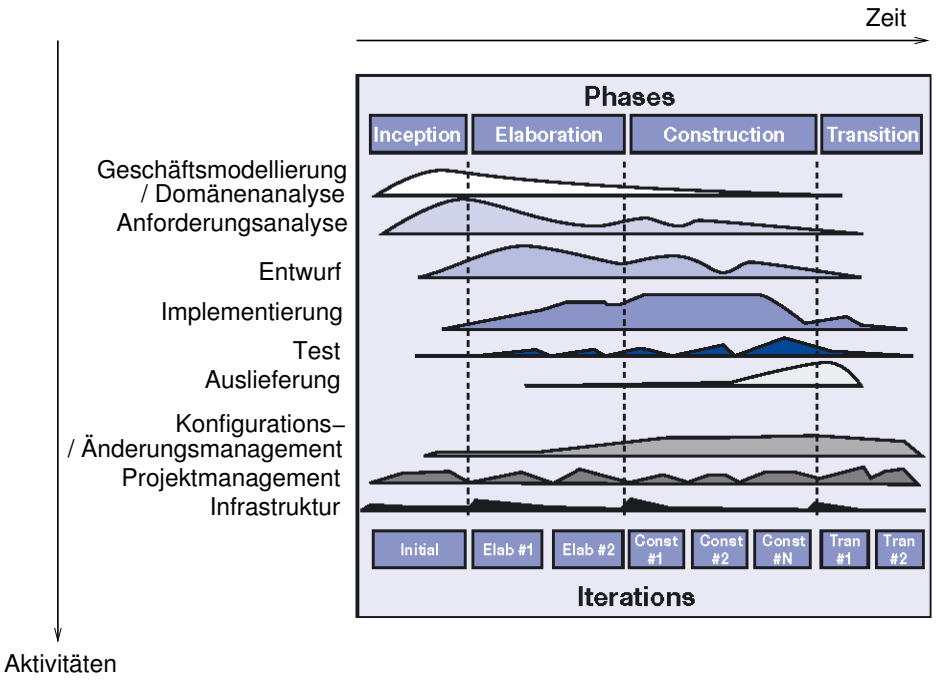
– schwierige Planung (was jedoch dem Problem inhärent ist)

– setzt große Flexibilität in der Organisation und beim Kunden voraus

Rational Unified Process (RUP) nach Gornik (2001)



Rational Unified Process (RUP) nach Gornik (2001)



Der Rational Unified Process (RUP) – als Konkretisierung des Unified Process der Firma Rational – ist ein weiteres inkrementelles Modell.
Der Prozess insgesamt gliedert sich in vier Phasen mit einer unterschiedlichen Anzahl von Iterationen. Das Produkt jeder Phase unterscheidet sich von den Produkten anderer Phasen.
Die Konzeptionsphase (Inception) erarbeitet die Anforderungen. Die Elaborationsphase (Elaboration) erstellt einen Entwurf. Die Konstruktionsphase (Construction) erstellt das implementierte System. Die Übergabephase (Transition) führt das System beim Kunden ein.
Jede Iteration gliedert sich in die Aktivitäten Geschäftsmodellierung, Anforderungsanalyse, Entwurf, Implementierung, Test und Auslieferung. Jede Iteration wird durch ein formales Review abgeschlossen.
Die Ausprägung der einzelnen Aktivitäten ist phasenabhängig. In der Konzeptphase z.B. dient eine Implementierung lediglich einem Konzeptbeweis (Machbarkeit kritischer Anforderungen) oder einer Demonstration (ein Benutzerschnittstellenprototyp). Es genügt hierfür eine weniger aufwändige, prototypische Implementierung (der Prototyp sollte anschließend weggeworfen werden!).
Der Aufwand jeder Aktivität variiert also in den Phasen. Dies wird durch die Kurven im Schaubild veranschaulicht. Die Geschäftsmodellierung beispielsweise erzeugt in der Konzeptionsphase naturgemäß einen hohen Aufwand, hat in der Übergabephase aber keine Bedeutung mehr. Alle Flächen gemeinsam ergeben den Gesamtaufwand des Projekts. Die Summe der Flächen pro Spalte ist der Aufwand pro Iteration. Die Summe der Flächen pro Zeile ist der Aufwand pro Aktivität.
Die Hauptaktivitäten sind Geschäftsmodellierung (optional), Anforderungsanalyse, Entwurf, Implementierung, Test, Auslieferung. Die Geschäftsmodellierung dient dazu, eine gemeinsame Sprache für die unterschiedlichen Gruppen Softwareentwickler und Betriebswirte zu finden und die Software-Modelle auf die zugrunde liegenden Geschäfsmodelle zurückzuführen. Die Geschäftsprozesse werden durch so genannte Geschäfts-Anwendungsfälle dokumentiert. Sie zielen darauf ab, ein gemeinsames Verständnis, welche Geschäftsprozesse in der Organisation unterstützt werden sollen, aller Beteiligten zu erreichen. Die Geschäfts-Anwendungsfälle werden analysiert, um zu verstehen, wie die Geschäftsprozesse unterstützt werden sollen.

RUP: Konzeptionsphase (Inception)

Ziel: "Business-Case" erstellen und Projektgegenstand abgrenzen.
Resultate:

- Vision der Hauptanforderungen, Schlüsselfeatures und wesentliche Einschränkungen
- initiale Anwendungsfälle (10-20% vollständig)
- Glossar oder auch Domänenmodell
- initialer Business-Case: Geschäftskontext, Erfolgskriterien (Schätzung des erzielten Gewinns, Marktanalyse etc.) und Finanzvorschau
- initiale Risikobetrachtung
- Projektplan mit Phasen und Iterationen
- Business-Modell falls notwendig
- ein oder mehrere Prototypen

Begleitende Aktivitäten sind das Konfigurations- und Änderungsmanagement und das Projektmanagement. Ihr Aufwand ist in allen Phasen mehr oder minder gleich. Der Aufwand für das Konfigurations- und Änderungsmanagement zeigt leichte Peaks im Übergang von einer Phase zur anderen, wenn die Konfigurationen fest gezurrt werden und zum Teil nachgearbeitet werden muss.
Dem Glossar kommt eine ganz besondere Bedeutung zu. Es erklärt die Begriffe der Anwendungsdomäne. Software-Entwickler sind Spezialisten für die Entwicklung von Software, aber Laien in sehr vielen ihrer Anwendungsdomänen. Darüber hinaus verwenden auch Kunden oft die Begriffe uneinheitlich bzw. geläufige Worte mit einer ganz speziellen Bedeutung in ihrem Kontext. Mißverständnisse zwischen Kunden und Softwareentwickler sind sehr häufig und können zu teuren Fehlentwicklungen führen.
Eine Marssonde, bei deren Entwicklung europäische und amerikanische Organisationen mitwirkten, verfehlte ihr Ziel, weil den Organisationen nicht bewusst war, dass sie unterschiedliche metrische Systeme für ihre Software zugrunde legten. Die einen interpretierten einen Wert in Zentimetern, die anderen in Zoll (Inch).
Im Glossar beschreibt der Software-Entwickler, was die Begriffe des Kunden bedeuten, ebenso wie seine eigenen speziellen Begriffe. Der Kunde begutachtet das Glossar. Damit definieren beide Partein ihr Vokabular. Mißverständnisse sollen so minimiert werden.
Das Domänenmodell (oft auch konzeptuelles Modell genannt) beschreibt die Begriffe/Objekte der Anwendungsdomäne und ihre Relationen.
Eine Reihe der genannten Punkte wird sicherlich in Zusammenarbeit mit Betriebswirten ausgearbeitet. Softwareentwickler haben eine Liebe zur Technologie, übersehen jedoch leider oft die Wirtschaftlichkeit ihrer Ideen. Mit ihr steht und fällt jedoch das Projekt.
Die Erstellung von Prototypen hat das Ziel, mögliche technologische Risiken auszuschließen, dem Kunden ein konkretes Bild der möglichen Anwendung zu vermitteln (Benutzerschnittstellenprototyp), Anforderungen zu konkretisieren ("I know it when I see it") und die Machbarkeit bestimmter Anforderungen zu demonstrieren.
Das Business-Modell erläutert, wie das System eingesetzt wird, um damit Profite zu erzielen.

RUP: Elaborationsphase (Elaboration)

Ziel: Verständnis der Anwendungsdomäne, tragfähige Software-Architektur, Projektplan, Eliminierung der Risiken

- Anwendungsfallmodell (mind. 80% fertig)
  - alle Anwendungsfälle und Akteure sind identifiziert,
  - die meisten Anwendungsfallbeschreibungen wurden entwickelt
- zusätzliche nichtfunktionale Anforderungen und Anforderungen, die nicht mit einem spezifischen Anwendungsfall assoziiert sind
- Beschreibung der Software-Architektur
- ausführbarer Architekturprototyp

Die Elaborationsphase ist die kritischste Phase. Hier entscheidet sich, ob das System tatsächlich gebaut wird. Der Engineering-Anteil ist weitgehend erbracht. Bis dahin halten sich die Kosten noch in Grenzen. Nun schließen sich die teure Konstruktions- und Übergabephase an.
Die Aktivitäten der Elaborationsphase stellen sicher, dass die Software-Architektur, die Anforderungen und Pläne hinreichend stabil sind (mögliche Änderungen sind antizipiert, völlig ausschließen lassen sie sich meist nicht) und Risiken sind ausreichend betrachtet, so dass Kosten und Zeitplan zuverlässig geschätzt werden können. Ab hier sollte man sich auf eine Projektdurchführung mit festem Preis einlassen können.
In der Elaborationsphase wird ein ausführbarer Architekturprototyp in ein oder mehr Iterationen erstellt. Die Anzahl der Iterationen hängt vom Scope, der Größe, der Risiken und dem Grad des Unbekannten des Projekts ab. Zumindest die kritischen Anwendungsfälle sollten hierfür einbezogen werden, da sie typischerweise die größten technischen Risiken aufwerfen. Ein evolutionärer Prototyp (d.h. einer der schrittweise ausgebaut wird) kann durchaus verwendet werden. Man sollte jedoch auch einige explorative Wegwerfprototypen in Erwägung ziehen, um spezifische Risiken wie z.B. Entwurfs- oder Anforderungskompromisse auszuloten. Sie dienen auch als Machbarkeitsstudien und Demonstrationen.

RUP: Elaborationsphase (Elaboration); Fortsetzung

- überarbeitete Liste der Risiken und überarbeiteter Business-Case
- Plan für das gesamte Projekt sowie grober Plan für die Iterationen und Meilensteine
- ein vorläufiges Benutzerhandbuch (optional)

Die Erstellung des Benutzerhandbuchs kann bereits frühzeitig beginnen. Es ist konkreter als die Anforderungsspezifikation und abstrakter als die Implementierung. Somit kann es als Brücke von den Anforderungen zur Implementierung benutzt werden.
Das vorläufige Handbuch dient sowohl als Spezifikation für die Implementierung und den Test als auch für die intensivere Auseinandersetzung mit der Benutzerführung (Beispiel: Was orthogonal und einfach zu beschreiben ist, kann oft auch orthogonal und einfach implementiert werden). Überlegungen zur Benutzerführung sollten frühzeitig gemacht werden, weil Änderungen größere Restrukturierungen nach sich ziehen können.

RUP: Konstruktionsphase (Construction)

Ziel: Fertigstellung, Integration und Test aller Komponenten; auslieferbares Produkt.

- Software-Produkt integriert in die entsprechende Plattform
- Benutzerhandbuch
- Dokumentation des gegenwärtigen Releases

In der Konstruktionsphase werden alle übrigen Komponenten und Feature realisiert und in das Produkt integriert und intensiv getestet. Die Konstruktionsphase ist einem gewissen Sinne ein Herstellungsprozess, bei dem Wert auf das Management von Ressourcen und das Controlling gelegt wird, um Kosten, Zeitplan und Qualität zu optimieren. In diesem Sinne geht der Prozess nun von der intellektuellen Entwicklung zur Entwicklung auslieferbarer Produkte über.
Viele Projekte sind groß genug, um die Konstruktion zu parallelisieren. Die Parallelisierung kann die Verfügbarkeit auslieferbarer Releases beschleunigen. Andererseits kann sie auch die Komplexität der Ressourcenverwaltung und der Synchronisation der Arbeitsflüsse erhöhen.
Eine robuste Architektur und ein verständlicher Plan hängen stark zusammen. Deshalb ist eine kritische Qualität der Architektur die Einfachheit ihrer Konstruktion. Dies ist einer der Gründe, weshalb die ausgeglichene Entwicklung der Architektur und des Plans während der Elaborationsphase so sehr betont wird.
Das Resultat der Konstruktionsphase ist ein Produkt, das tatsächlich in die Hände des Benutzers übergehen kann.

RUP: Übergabephase (Transition)

Ziel: Produkt wird der Benutzergemeinde übergeben.
Hauptziele im Einzelnen:

- Benutzer sollten sich möglichst alleine zurecht finden.
- Beteiligte sind überzeugt, dass die Entwicklungs-Baselines vollständig und konsistent mit den Evaluationskriterien für die Vision sind.
- Erreichung der letzten Produkt-Baseline so schnell und kostengünstig wie möglich.

RUP: Übergabephase (Transition)

Typische Tätigkeiten:

- "Beta-Test", um das neue System gegen die Benutzererwartungen zu validieren
- Parallele Verwendung mit einem Legacy-System, das durch das Produkt ersetzt werden soll
- Konversion aller notwendigen Daten (Dateien und Datenbanken)
- Schulung aller Benutzer und Administratoren
- Übergabe an Marketing, Vertrieb und Verkäufer

Wenn ein Produkt ausgeliefert wird, ergibt sich in der Regel schnell die Notwendigkeit neuer Releases, um Probleme zu beseitigen, Features zu realisieren, deren Implementierung verschoben werden musste, und Erweiterungen vorzunehmen.
Die Übergabephase beginnt, wenn eine Baseline reif genug ist, um beim Endbenutzer installiert werden zu können. Das erfordert typischerweise, dass zumindest eine nützliche Teilmenge des Systems mit einer aktzeptablen Qualität fertig gestellt werden konnte und dass die Benutzerdokumentation vorhanden ist.
Meist fallen mehrere Iterationen in der Übergabephase an: Beta-Releases, allgemeine Releases, Bug-Fix- und Erweiterungsreleases. Hoher Aufwand wird in die Entwickler der benutzerorientierten Dokumentation, die Schulung von Benutzern, Unterstützung der Benutzer in ihren ersten Verwendungen des Produkts und Reaktion auf Benutzer-Feedback investiert.
Man sollte sich an diesem Punkt jedoch auf den Feinschliff, die Konfiguration, Installation und Usability-Aspekte beschränken. Gänzlich neue Erweitungen sollten durch einen nachfolgenden separaten Entwicklungszyklus realisiert werden.
Die Übergabephase kann trivial sein (Software und Handbuch wird auf einen Server im Internet gelegt) oder auch sehr aufwändig und kompliziert (Ersetzung einer Flugaufsichts-Software).

Empfohlene Anzahl von Iterationen nach Kruchten (1998)

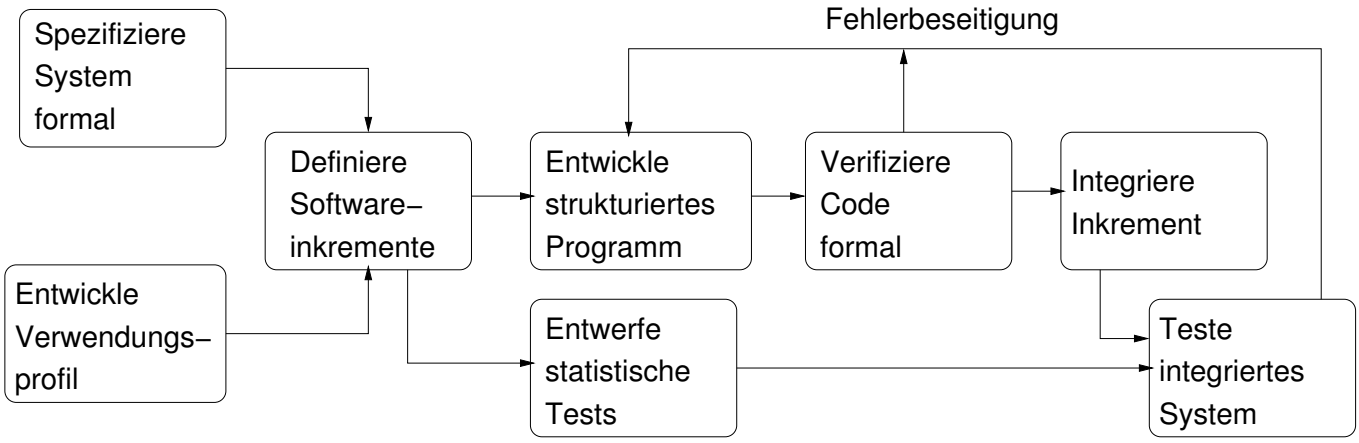
Komplexität	niedrig	normal	hoch
Konzeption	0	1	1
Elaboration	1	2	3
Konstruktion	1	2	3
Übergabe	1	1	2
Summe	3	6	9

Bewertung des RUPs

- Übernimmt vom Spiralmodell die Steuerung durch Risiken
- Konkretisiert die Aktivitäten (Spiralmodell ist ein Meta-Modell)
- Änderungen der Anforderungen sind leichter einzubeziehen als beim Wasserfallmodell
- Projekt-Team kann im Verlauf hinzulernen
- (Hauptsächlich) Konstruktionsphase kann inkrementell ausgestaltet werden

Iterativ ist nicht gleich inkrementell. Bei der iterativen Entwicklung werden Entwicklungsschritte wiederholt ausgeführt. Bei der inkrementellen Entwicklung geschieht dies auch, jedoch immer um eine neues Release auf Basis eines vorherigen zu bauen. Letzteres ist im Begriff Iteration nicht eingeschlossen.

Cleanroom Development (Mills u. a. 1987)



Cleanroom Development

Schlüsselstrategien

- Formale Spezifikation
- Inkrementelle Entwicklung
- Strukturierte Programmierung
- Statische Verifikation
- Statistisches Testen
  - basiert auf Verwendungsprofilen (die Verwendungsweise der Software in der Praxis)
  - die häufigsten (und kritischsten) Verwendungsarten werden verstärkt getestet

Statistisches Testen gleicht einem statistischen Experiment. Aus der formalen Spezifikation und den im Feld ermittelten Benutzungsprofilen werden geeignete Testfälle entwickelt. Mit Hilfe der Ergebnisse des Tests wird dann die Mean-Time-To-Failure (durschnittliche Dauer bis zu einem Störfall) mit einer gewissen Wahrscheinlichkeit vorhergesagt.

Cleanroom Development

Gruppen:

- Spezifikationsteam:
  - verantwortlich für Entwicklung und Wartung der Systemspezifikation
  - erstellt kundenorientierte und formale Spezifikation
- Entwicklungsteam:
  - verantwortlich für Entwicklung und Verifikation der Software
  - Software wird nicht ausgeführt hierzu!
  - verwendet Code-Inspektion ergänzt durch Korrektheitsüberlegungen (nicht streng formal)
- Zertifizierungsteam:
  - verantwortlich für statistische Tests

Cleanroom Development

Erfahrungen Cobb und Mills (1990):

- weniger Fehler als bei traditioneller Entwicklung
- bei vergleichbaren Kosten

Agiles Manifest

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

— `http://agilemanifesto.org/`

Prinzipien der agilen Entwicklung

- Kundenzufriedenheit durch schnelle Auslieferung nutzbarer Software
- funktionsfähige Software wird häufig ausgeliefert (eher Wochen als Monate)
- funktionsfähige Software ist das Maß für Fortschritt
- auch späte Änderungen der Anforderungen sind willkommen
- enge tägliche Kooperation von Geschäftsleuten und Entwicklern
- Konversation von Angesicht zu Angesicht ist die beste Kommunikationsform (gemeinsamer Ort)
- Projekte bestehen aus Menschen, denen man vertrauen sollte
- kontinuierliches Streben nach technischer Exzellenz und gutem Design
- Einfachheit
- selbstorganisierte Teams
- kontinuierliche Anpassung an sich ändernde Bedingungen

— `http://www.agilemanifesto.org/principles.html`

Varianten der agilen Entwicklung

- Agile Unified Process (AUP)
- Dynamic Systems Development Method (DSDM)
- Essential Unified Process (EssUP)
- **Extreme Programming (XP)**
- Feature Driven Development (FDD)
- Open Unified Process (OpenUP)
- **Scrum**

Extreme Programming (Beck 2000)

Extreme Programming (XP) ist eine agile Methode für

- kleinere bis größere Entwicklerteams (max. 10-15 Personen),
- Probleme mit vagen Anforderungen
- und Projekte, bei denen ein Kundenrepräsentant stets greifbar ist.

`http://www.extremeprogramming.org/`

Extreme Programming (Beck 2000)

Anerkannte Prinzipien und Praktiken werden „extrem“ umgesetzt:

- Code-Reviews → permanente Reviews durch Pair-Programming
- Testen → ständiges Testen: Unit-Tests sowie Akzeptanztests durch den Kunden/Benutzer
- klare Struktur → jeder verbessert sie kontinuierlich durch Refactoring
- Einfachheit → stets die einfachste Struktur wählen, die die aktuellen Anforderungen erfüllt
- Integration → permanente Integration auch mehrmals am Tag
- Validierung:
  - Kunde/Benutzer ist stets involviert bei der Planung neuer Iterationen und verantwortlich für Akzeptanztest
  - kurze Iterationen → Dauer in Minuten und Stunden, nicht Wochen, Tage, Jahre

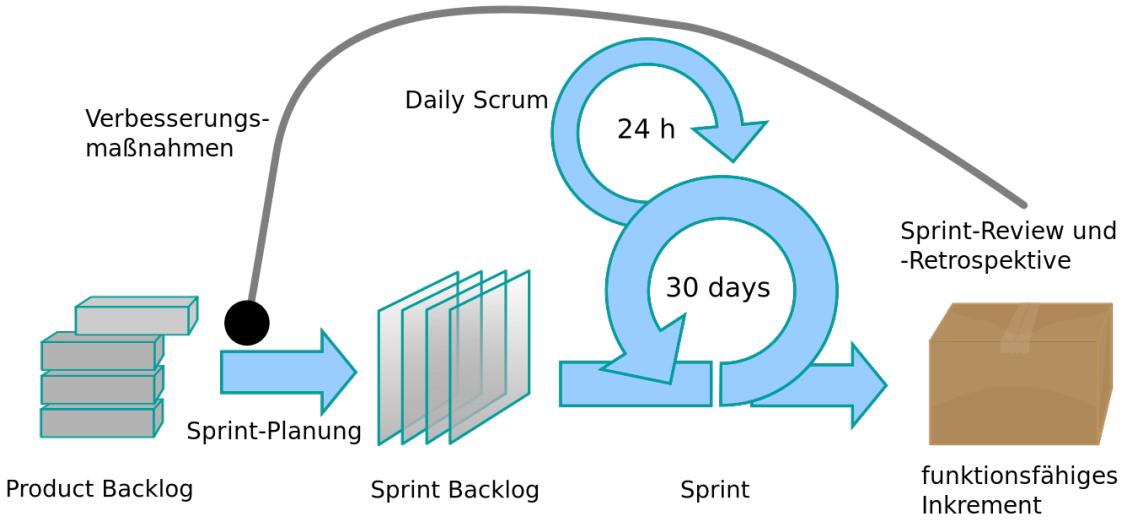
aber auch Auslassung anerkannter Prinzipien:

- Dokumentation: mündliche Überlieferung, Tests und Quellcode
- Planung: sehr begrenzter Horizont

Weitere XP-Charakteristika

- Kunde vor Ort
- eine Metapher statt einer Architekturbeschreibung
- 40-Stundenwoche
- Code ist kollektives Eigentum
- Kodierungsstandards

Scrum-Prozess



Bildquelle: http://en.wikipedia.org/wiki/File:Scrum_process.svg, GPL

Scrum-Rollen (Pichler 2008)

Product Owner:

- vertritt Endkundenbedürfnisse
- vereint Produkt- und Projektmanagementaufgaben
- fest integriert in Entwicklung

Aufgaben:

- Anforderungsbeschreibung und -management
- Releasemanagement und Return on Investment
- enge Zusammenarbeit mit dem Team
- Stakeholder-Management

Scrum-Rollen (Pichler 2008)

Team:

- entscheidet selbst, wieviele Anforderungen in einem Inkrement umgesetzt werden
- legt Arbeitsschritte und Arbeitsorganisation selbst fest
- agiert autonom
- interdisziplinär besetzt
- selbstorganisiert
- klein
- Vollzeitmitglieder
- Arbeitsplätze in unmittelbarer Nähe

Scrum-Rollen (Pichler 2008)

Scrum-Master (vom Team bestimmt):

Aufgaben

- etabliert Scrum
- unterstützt Team
- stellt Zusammenarbeit von Team und Product Owner sicher
- beseitigt Hindernisse
- verbessert Entwicklungspraktiken
- führt durch dienen

Fähigkeiten

- Moderation
- Coaching
- Erfahrung in Softwareentwicklung

Scrum: Product Backlog

Product Backlog enthält

- alle bekannten funktionalen und nichtfunktionalen Anforderungen
- weitere Arbeitsergebnisse (z.B. Aufsetzen der Test- und Entwicklungsumgebung)

keine Aktivitäten!

wird vom Product Owner gepflegt

Scrum: Product Backlog

Prio	Thema	Beschreibung	Akzeptanz	Aufwand
1	Kalender	Administrator kann zentrale Kalenderdatenbank anlegen	Installationsskript legt DB an	2
2	Kalender	Nutzer kann Termin eintragen	valider Termin wird eingetragen, invalider Termin wird zurückgewiesen	3
3	Kalender	Nutzer kann Termin löschen	gelöschter Termin ist entfernt; Löschung nur, wenn Rechte vorhanden	3
...	...	...	...	...

Das Product Backlog ist ein lebendes Dokument. Es wird nach jedem Sprint aktualisiert. Seine Einträge sind priorisiert. Sie weisen einen unterschiedlichen Detaillierungsgrad auf: Die im kommenden Sprint anvisierten Anforderungen sind genauer beschrieben. Der Aufwand jedes Eintrags ist abgeschätzt (Einheit: Punkte; siehe unten).
Vorlagen gibt es z.B. unter
`http://epf.eclipse.org/wikis/scrum/Scrum/guidances/templates/burndown_chart_D182CF23.html`

Scrum: Kostenschätzung

Schätzklausur und Planungspoker

Scrum: Kostenschätzung

Punkteskala (Fibonacci-Reihe):

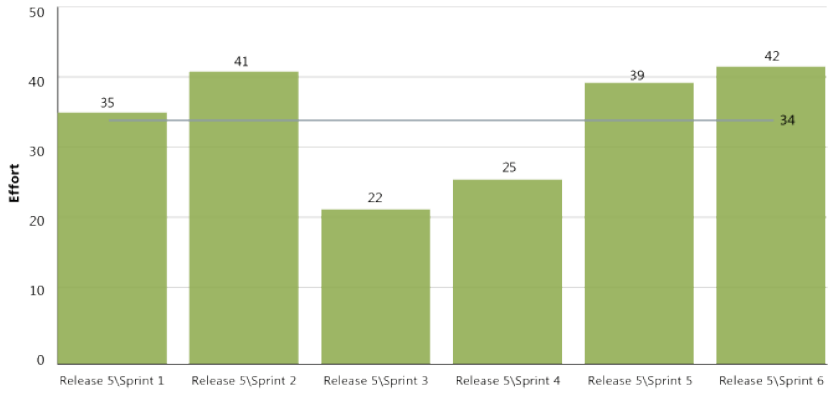
0	kein Aufwand
1	sehr kleiner Aufwand
2	kleiner Aufwand = 2 × sehr kleiner Aufwand
3	mittlerer Aufwand = sehr kleiner + kleiner Aufwand
5	großer Aufwand = kleiner + mittlerer Aufwand
8	sehr großer Aufwand = mittlerer + großer Aufwand
13	riesiger Aufwand = großer + sehr großer Aufwand

Scrum: Entwicklungsgeschwindigkeit

Punkte werden im Projektverlauf auf Echtzeit abgebildet

Entwicklungsgeschwindigkeit (Velocity) = Punkte / Sprint

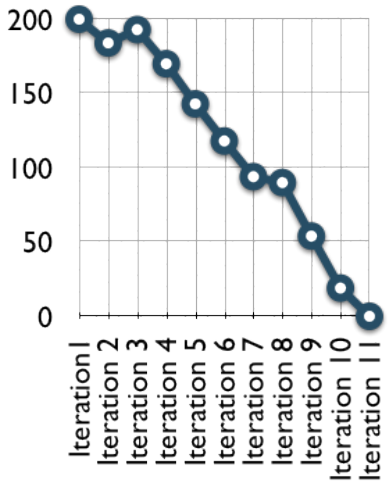
Velocity Chart



Scrum: Steuerungselemente

Sprint Burndown Chart

Release Burndown Chart



Agile versus weit voraus planende Prozessmodelle

Risiken agiler Methode:

- Skalierbarkeit, Kritikalität, Einfachheit des Entwurfs, Personalfluktuation, Personalfähigkeiten

Risiken weit voraus planender Prozessmodelle:

- Stabilität der Anforderungen, steter Wandel, Notwendigkeit schneller Resultate, Personalfähigkeiten

Generelle Risiken:

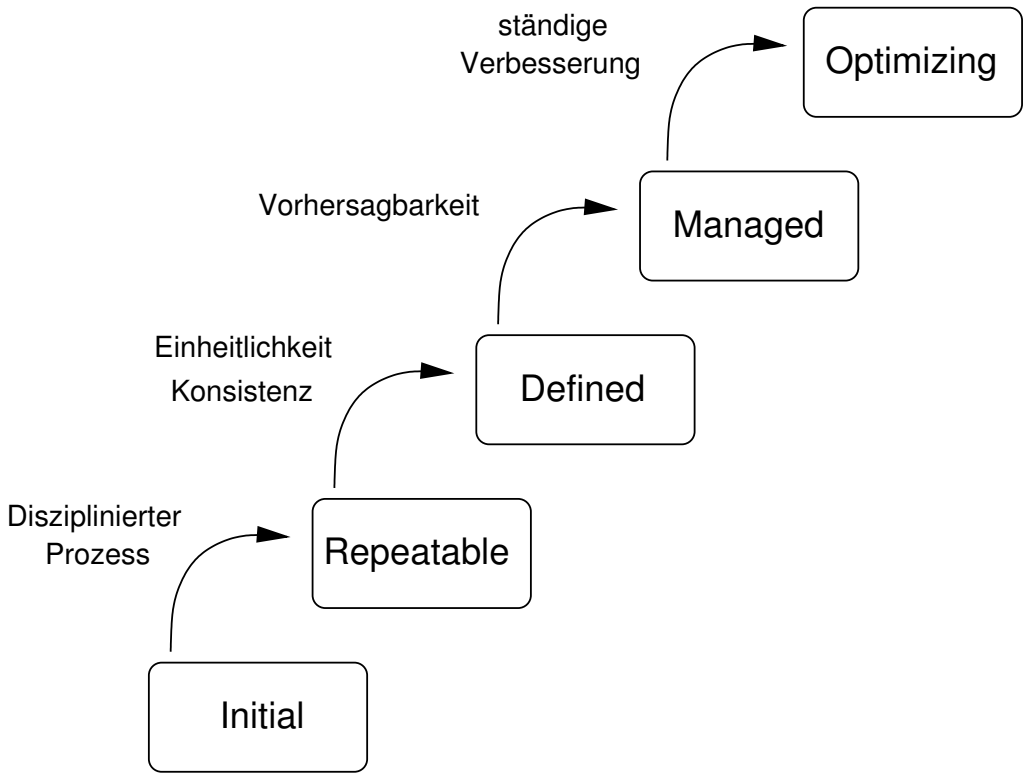
- Unsicherheiten bei der Technologie, unterschiedliche Interessengruppen, komplexe Systeme

— Boehm und Turner (2003)

Capability Maturity Model

- Entwickelt vom SEI 1985-91 für DoD
- Beschreibt Stufen der Prozessreife
- Maßstab/Leitfaden für Verbesserungen
- Idee: besserer Prozess $\rightarrow$ besseres Produkt
- 5 Stufen (CMM Level 1-5)
- Definiert Schlüsselbereiche (Key Process Areas)
- Steigende Transparenz des Prozesses

Capability Maturity Model



CMM Level 1 – Initial

- Prozess meist völlig instabil
- Typisch: in Krisen nur noch Code & Fix
- Qualität und Fertigstellung unvorhersagbar
- Erfolge nur durch gute Leute und großen Einsatz

CMM Level 2 – Repeatable

- Projekterfolge nachvollziehbar und wiederholbar
- Projektplanung und -management basiert auf Erfahrung
- Key Process Areas:
  - Anforderungsverwaltung (u.a. Reviews)
  - Projektplanung (Zeitplanung, Risikomgmt., Prozess)
  - Projektverfolgung und -überblick (Transparenz)
  - Unterauftragsverwaltung
  - Qualitätssicherung (QS-Plan)
  - Konfigurationsverwaltung (Konsistenz, Änderungsverfolgung)

CMM Level 3 – Defined

- Organisationsweiter Prozess, wird für jedes Projekt angepasst
- Zuständig: Software Engineering Process Group
- Kosten, Zeitplan und Funktionalität im Griff
- Key Process Areas:
  - Organisationsweiter Prozess
  - Prozessdefinition
  - Ausbildungsprogramm
  - Integriertes Softwaremanagement (Anpassung auf konkretes Projekt)
  - Software-Engineering-Techniken, Tool-Unterstützung
  - Koordination zwischen Gruppen
  - Peer Reviews

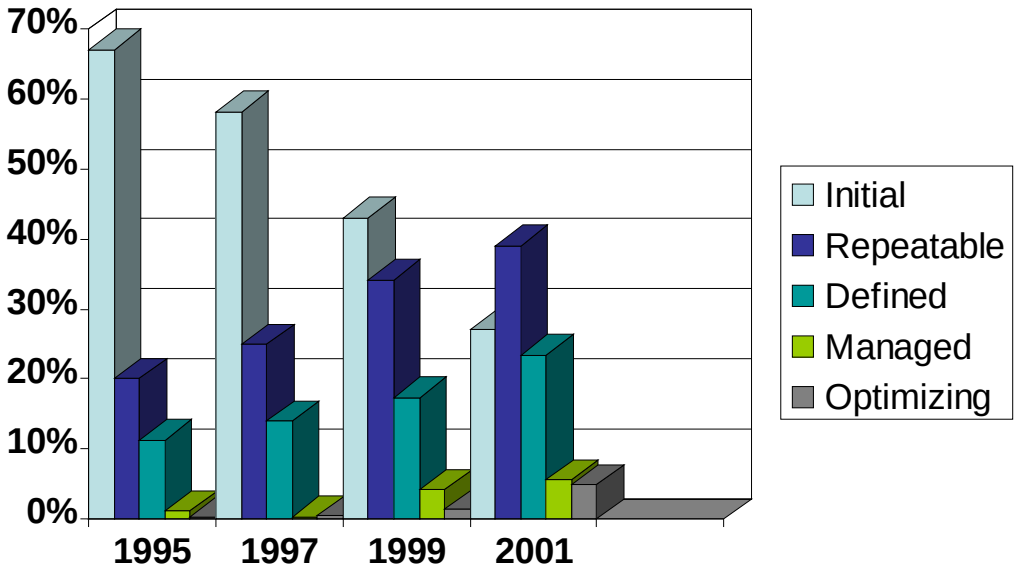
CMM Level 4 – Managed

- Einbeziehung quantitativer Aspekte
- Ziele setzen und überwachen
- Prozessmessdaten werden aufgenommen, archiviert, analysiert
- Vorhersagbarkeit steigt
- Key Process Areas:
  - Quantitative Prozesssteuerung
    → Leistung des Prozesses überwachen
  - Software-Qualitätsmanagement
    → Messbare Ziele für Prozessqualität

CMM Level 5 – Optimizing

- Änderungsmanagement für Technologie und Prozesse
- Feedback von Projekten zum Prozess → ständige Verbesserung
- Key Process Areas:
  - Defektvermeidung
    → Analyse von Fehler-/Problemursachen
    → Vermeiden erneuten Auftretens
  - Verwaltung von Technologieänderung
    → neue Technologien bewerten, evtl. einführen
  - Verwaltung von Prozessänderung
    → kontinuierliche Verbesserung

Capability Maturity Model

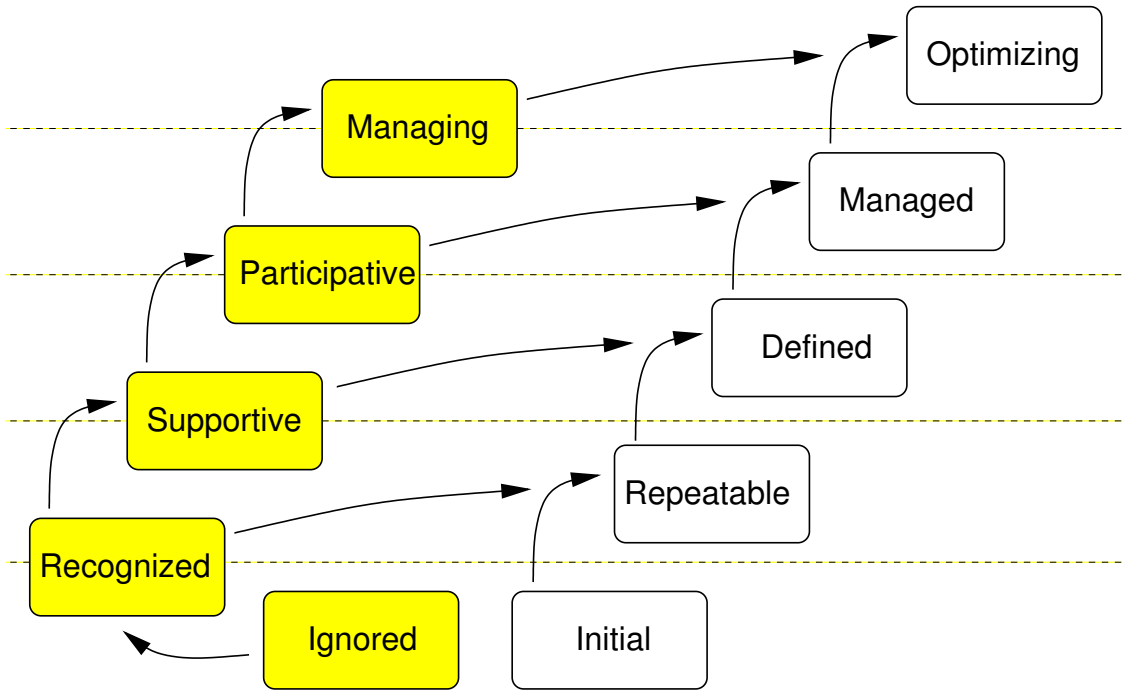


SEI Assessments (Quelle: SEI)

Probleme mit CMM

- Management: „zu teuer“
- Wenig verbreitet (ca. 10-15%)
- Nur langsame Verbesserung (ca. 2 Jahre/Level)
- Neue Technologien nicht berücksichtigt

Capability Maturity Model – Management



Quelle: Parker (1995)

Capability Maturity Model Integration (CMMI)

- Viele Maturity-Model-Varianten

→ CMMI: „ Integration“ (SEI 2000):

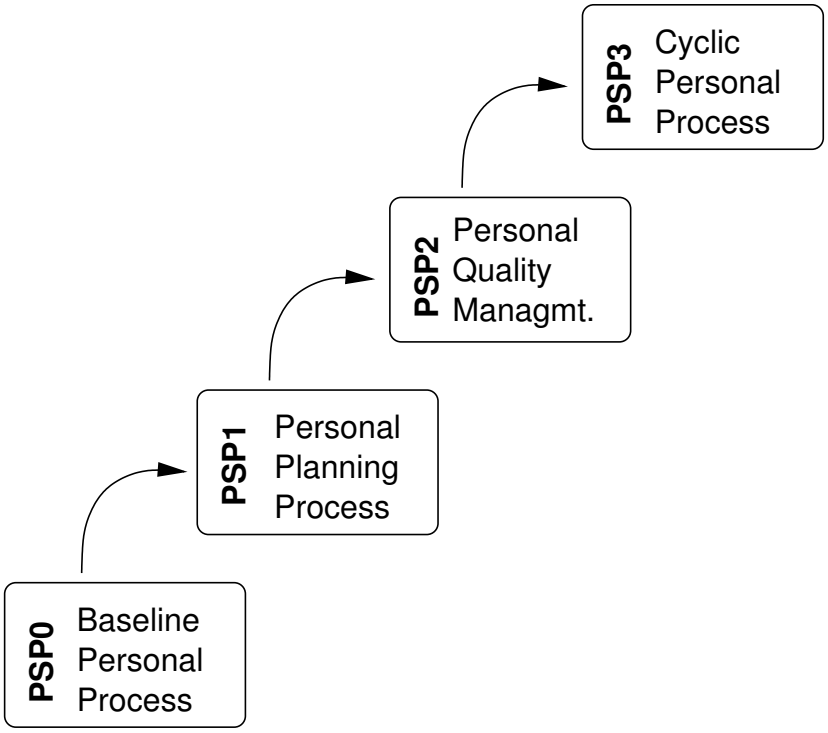
- Angepasst an iterative Entwicklung
- Generische Ziele hinzugefügt
- Zusätzliche KPAs:
  - Level 2: Measurement and Analysis
  - Level 3: Software Product Engineering (verfeinert); Risk Management; Decision Analysis and Resolution
  - Level 4, 5: nur Restrukturierungen

Persönlicher Softwareprozess (PSP)

- Watts S. Humphrey (1995) (SEI)
- Anwendung von CMM auf einen Entwickler
- Schwerpunkte: Planung, Qualität
- Vorteile:
  - Schneller umsetzbar
  - Konkrete Techniken angebbar
  - Verbesserungen sofort wahrnehmbar

  → höhere Motivation

PSP – Evolution



PSP0 – Baseline Personal Process

PSP0: Bisherige Vorgehensweise plus

- Messung
  - Zeit pro Phase
  - gefundene/gemachte Fehler pro Phase
  - Zeit für Fehlerbehebung
- Formulare: Projektplan, Zeiten, Fehler

PSP0.1: plus

- Codierrichtlinien
- Messung der LOC (Veränderungen)
- Formular: Prozessverbesserung

PSP1 – Personal Planning Process

PSP1: PSP0.1 plus

- Größenschätzung mit PROBE (PROxy-Based Estimating)
  - Schätzung basiert auf Objekten (Entwurf)
  - Unterscheidet Objekte nach Typ, Größe, #Methoden
  - Daten sammeln, Regressionsanalyse
- Formular: Testbericht

PSP1.1: PSP1 plus

- Aufgabenplanung
- Zeitschätzung, -planung
- Projektverfolgung (Earned Value Tracking)

PSP2 – Personal Quality Management

PSP2: PSP1.1 plus

- Code Reviews (Checklisten)
- Design Reviews (Checklisten)

PSP2.1: PSP2 plus

- Design Templates:
  - Operational Scenario ($\approx$ Anwendungsfall)
  - Functional Specification (formale Spezifikation)
  - State Specification ($\approx$ Zustandsdiagramm)
  - Logic Specification (Pseudocode)

$\rightarrow$ Vermeidung von Designfehlern

$\rightarrow$ Beurteilung der Qualität

- Cost-of-Quality (Behebung, Bewertung, Vermeidung)

PSP3 – Cyclic Personal Process

PSP3:

- Anwendung auf große Projekte
- Nach High-Level Design: Aufteilung in Module
- Anwendung von PSP2.1 auf jedes Modul
- Formular: Issue tracking log

Wiederholungs- und Vertiefungsfragen

- Erläutern Sie die Ideen sowie Vor- und Nachteile der Entwicklungsprozesse...
  - Wasserfall
  - V-Modell
  - testgetriebene Entwicklung
  - inkrementelle Entwicklung
  - Spiralmodell
  - Rational Unified Process
  - Cleanroom Development
  - Extreme Programming
- Gegeben ist das folgende Szenario: [...]. Welches Vorgehensmodell empfehlen Sie?
- Unter welchen Umständen würden Sie eher agiles Vorgehen als voraus planendes empfehlen?
- Stellen Sie das Capability Maturity Model dar. Wozu dient es?
- Wie können Prozesse verbessert werden?
- Was ist der persönliche Software-Entwicklungsprozess? Wozu dient er?

Weiterführende Literatur

Bibliographie zu Entwicklungsprozessen:

Arbeitskreis der Fachgruppe 5.11
"Begriffe und Konzepte der Vorgehensmodellierung"
`http://www.vorgehensmodelle.de/giak/arbeitskreise/vorgehensmodelle/publallg.html`

Rational Unified Process: Kruchten (1998)

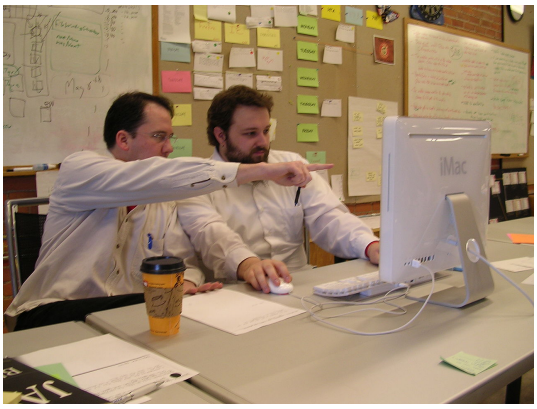
Testgetriebene Entwicklung und Paarprogrammierung

Testgetriebene Entwicklung und Paarprogrammierung
Pair-Programming
Testgetriebene Entwicklung
Aufgabe

Rollen beim Pair-Programming

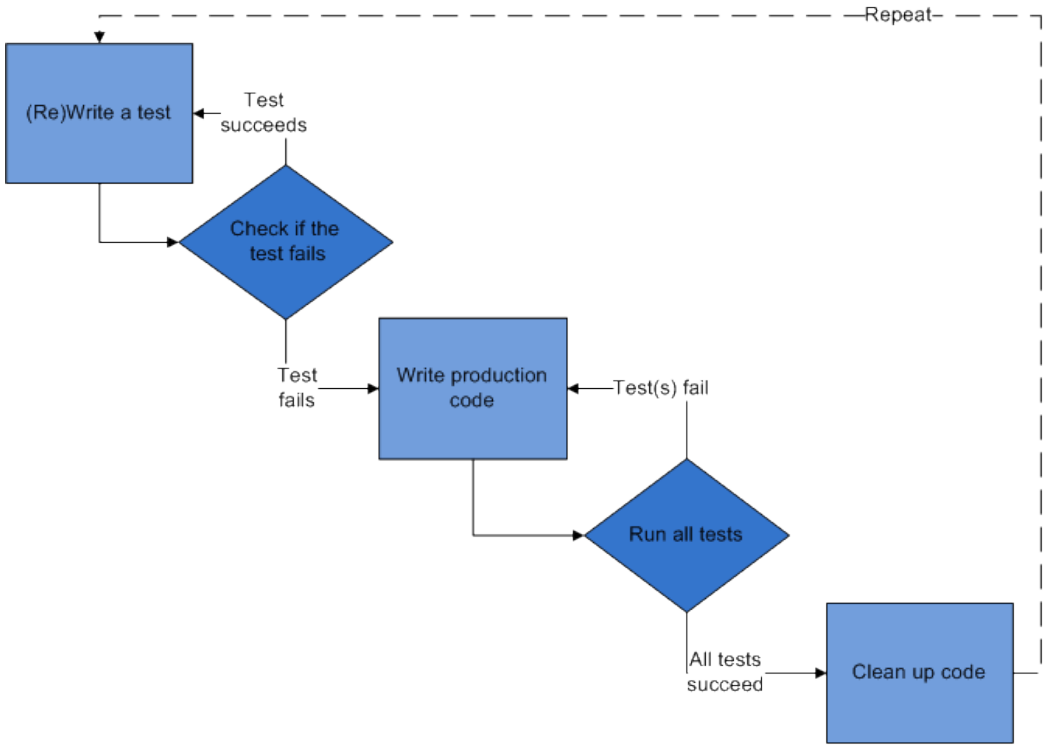
- **Fahrer (Driver)**
  - schreibt Code
  - bedenkt taktische Fragen
- **Beifahrer (Observer, Navigator)**
  - begutachtet geschriebenen Code
  - bedenkt strategische Fragen

Rollen werden häufig getauscht.



Quelle: `http://en.wikipedia.org/wiki/Pair_programming`

Testgetriebene Entwicklung



Quelle: `https://en.wikipedia.org/wiki/Test-driven_development`

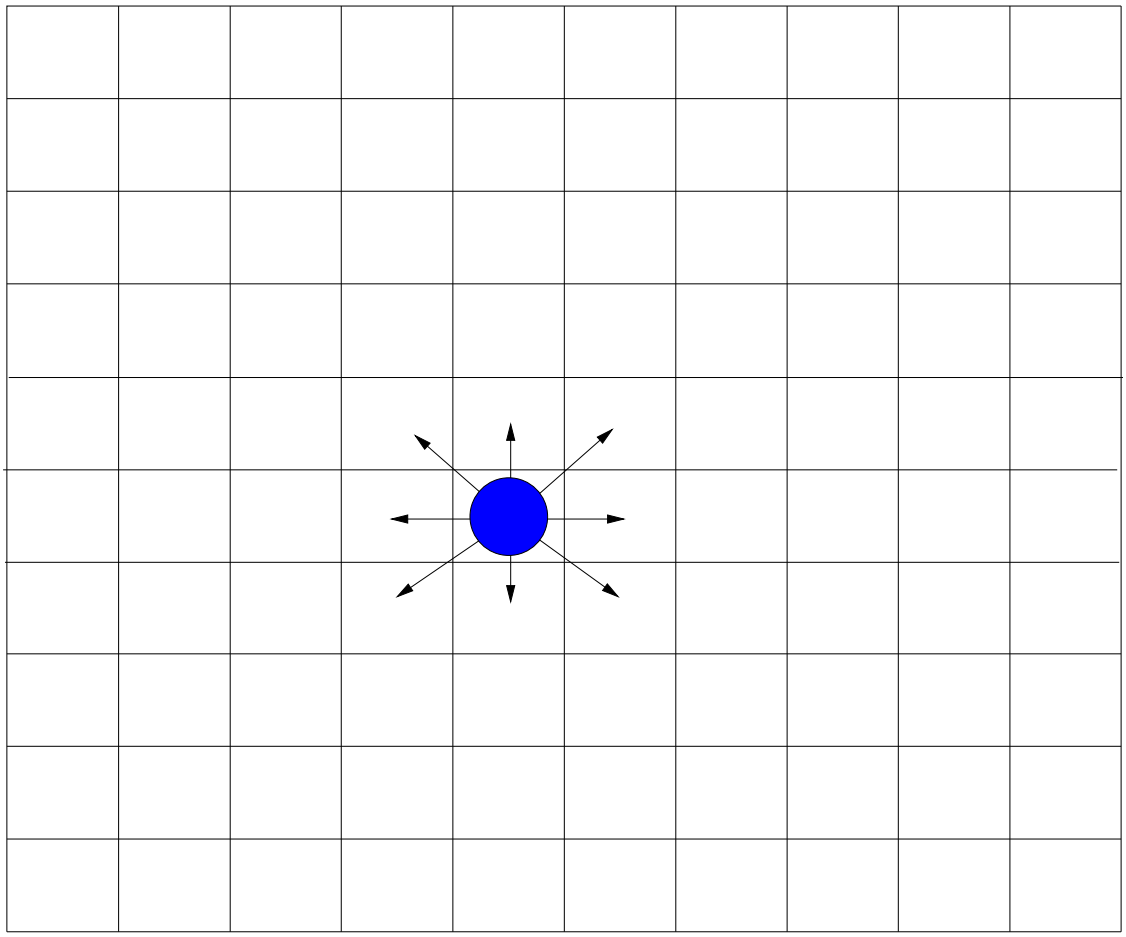
Conway's Game of Life

Eingabe:

- unendlich ausgedehnte zweidimensionale Matrix I der Generation i
- jede Zelle hat zwei Zustände:
  - enthält ein Lebewesen $\rightarrow$ lebendige Zelle
  - enthält kein Lebewesen $\rightarrow$ tote Zelle

Nachbarn eines Lebewesens in Zelle z: alle Lebewesen, die in einer der direkt angrenzenden Zellen oberhalb, unterhalb, rechts, links, schräg links oberhalb, schräg rechts oberhalb, schräg links unterhalb oder schräg rechts unterhalb von z leben.

Conway's Game of Life



Conway's Game of Life

Ausgabe:

- unendlich ausgedehnte zweidimensionale Matrix I' mit den Lebewesen der Generation $i + 1$
- I' ergibt sich wie folgt aus I (für jede Zelle $z \in I$):
  - lebendiges z mit weniger als zwei lebendigen Nachbarn stirbt (Unterbevölkerung)
  - lebendiges z mit zwei oder drei lebendigen Nachbarn überlebt
  - lebendiges z mit mehr als drei lebendigen Nachbarn stirbt (Überbevölkerung)
  - totes z mit genau als drei lebendigen Nachbarn beginnt zu leben (Reproduktion)

Kosten- und Aufwandsschätzung

Kosten- und Aufwandsschätzung
Kostenschätzung
Function-Points
Object-Points
COCOMO
Wiederholungsfragen

Kostenschätzung

Wichtige Fragen vor einer Software-Entwicklung:

- Wie hoch wird der Aufwand sein?
- Wie lange wird die Entwicklung dauern?
- Wie viele Leute werden benötigt?

Frühzeitige Beantwortung wichtig für:

- Kalkulation und Angebot
- Personalplanung
- Entscheidung „make or buy“
- Nachkalkulation

Schätzgegenstand

- **Kosten**: was zu bezahlen ist
  - Aufwand × Kosten pro Aufwandseinheit
  - Schätzen: 50 Euro/DLOC (Delivered Line of Code)
- **Dauer**: wie lange man warten muss
  - Aufwand in PM/Anzahl Mitarbeiter
    (reell nicht-linearer Zusammenhang laut Brooks (1995))
  - Parkinsons Gesetz
- **Aufwand**: was man an Ressourcen braucht
  - Umfang + Risiko + Management + ...

Ansätze

- Expertenschätzung
- Analogiemethode

- Top-Down-Schätzung
- Bottom-Up-Schätzung

- Pricing-to-Win/Festpreis

- Algorithmische Schätzung
  - COCOMO: Anzahl Codezeilen
  - Schätzung auf Basis von Function-Points: Ein- und Ausgaben

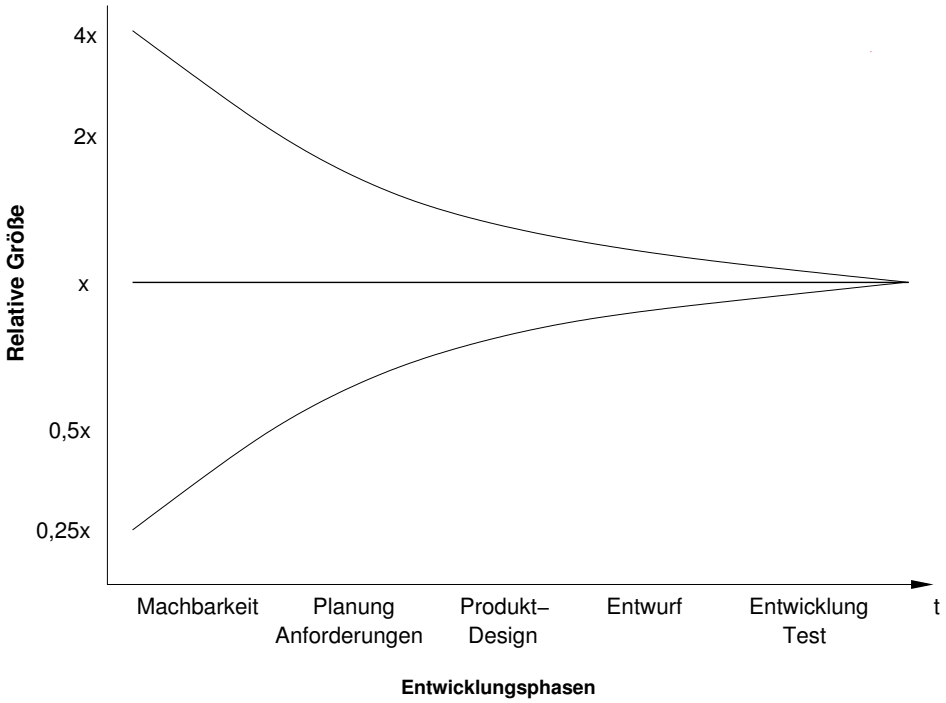
- Expertenschätzung: mehrere Experten fragen; Abweichungen diskutieren, bis Konsens erreicht
  → Schätzpoker bei Scrum
- Analogie: dauert so lange wie beim ähnlichen Projekt X; Bezug auf historische Daten eines ähnlichen Projekts

- Top-Down-Schätzung: Ableitung aus globalen Größen
  - z.B. Aufwand steht fest, daraus Umfang ableiten
- Bottom-Up-Schätzung: Dekomposition und Schätzung der einzelnen Komponenten sowie deren Integrationsaufwand

Pricing-To-Win: Preis wird vereinbart; im Zuge des Projekts einigt man sich auf Funktionsumfang (eigentlich keine Schätzung).
Berechnung aus früh bekannten Größen; statistisches Kostenmodell aus Vergangenheitsdaten wird erstellt; Modell wird zur Vorhersage benutzt.

Kostenschätzung

Boehm (1981)



Function-Point-Methode

Benötigt für frühe Kostenschätzung: Maß für den Umfang der Software.

Function-Points:

- Messen des **funktionalen Umfangs**[1] einer zu erstellenden Software aus Benutzersicht
- Eingabe: Lastenheft

Einsatz:

- Umrechnung des Umfangs in personellen Aufwand
- Vergleich und Bewertung von Systemen
- Messung von Produktivität, Benchmarking

[1] nicht des Aufwands

Historische Entwicklung der Function-Point-Methode

- 1979: erste Veröffentlichung: Alan J. Albrecht (1979) (IBM)
- 1985: Veröffentlichung der „IBM-Kurve“ (IBM 1985): Zusammenhang von Aufwand und Function-Points für 54 Projekte
- 1990: Hype: fast alle großen Unternehmen probieren Function-Point-Analyse aus
- 1995: Abkühlung des Interesses
- 1995-heute: wieder gesteigertes Interesse:
- Heute:
  - zahlreiche Varianten
  - IFPUG Int'l Function Point User Group

- 1979: erste Veröffentlichung: Alan J. Albrecht (1979) (IBM)
- 1985: Veröffentlichung der „IBM-Kurve“ (IBM 1985): Zusammenhang von Aufwand und Function-Points für 54 Projekte
- 1990: Hype: fast alle großen Unternehmen probieren Function-Point-Analyse aus
- 1995: Abkühlung des Interesses
  - Unerfahrenheit der Anwender
  - unrealistische Erwartungen
  - zu wenig Erfahrungsdaten vorhanden
- 1995-heute: wieder gesteigertes Interesse:
  - Benchmarking
  - Wirtschaftlichkeit (Function-Points versus Kosten)
  - Outsourcing
  - Offshoring
- Heute:
  - zahlreiche Varianten
  - IFPUG Int'l Function Point User Group

`http://www.ifpug.com/fpafund.htm`
FAQ: `http://ourworld.compuserve.com/homepages/softcomp/fpfaq.htm`

Function-Point-Methode – Vorgehen

1. Zähltyp festlegen: Neu-/Weiterentwicklung,
2. Systemgrenzen festlegen
3. Identifizieren der Elementarprozesse und deren Funktionstypen sowie der Datenbestände
4. Bewerten des Umfangs der Funktionstypen und Datenbestände
5. Ermittlung der gewichteten Function Points durch Schätzung von technischen Randbedingungen
6. Verwendung; z.B. Ermittlung des Aufwands

Elementarprozess

Definition

Elementarprozess: atomare und einzigartige Aktivität des Systems aus Benutzersicht

Definition

Atomarizitätsprinzip: Elementarprozess ist kleinste aus fachlicher Sicht sinnvolle, in sich abgeschlossene Aktivität

Bsp.: Erfassung einer Kundenadresse (auch über mehrere Bildschirmmasken)

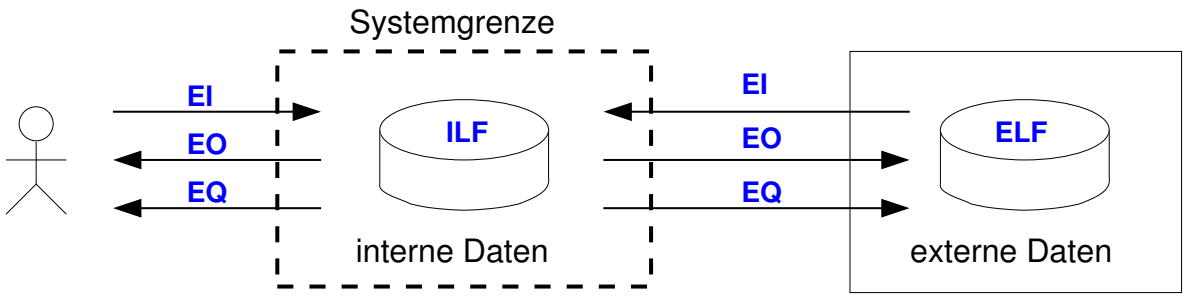
Definition

Einmaligkeitsprinzip: Elementarprozess gilt als einmalig (einzigartig), wenn er durch die ein- oder ausgegebenen Daten oder durch die Verarbeitungslogik unterscheidbar ist (aus Sicht des Anwenders); Unterscheidung durch

- besondere Verarbeitungslogik oder
- verarbeitete Felder der Datenbestände oder
- verarbeitete Datenbestände selbst

Bsp.: Berechnung des Krankenkassenbeitrags einer privaten Versicherung für Neu- bzw. Bestandskunden sind verschiedene Elementarprozesse

FP – Identifizieren der Funktionstypen



Funktionstypen von Elementarprozessen:

- Eingaben: EI (External Input; Eingabe für ILF)
- Ausgaben: EO (External Output; Ausgabe abgeleiteter Daten)
- Abfragen: EQ (External Inquiry; reine Abfrage von Daten ohne Berechnung/Ableitung)

Funktionstypen von Datenbeständen:

- Interner Datenbestand (ILF: Internal Logical File): fachliche Daten, die vom System selbst gepflegt werden (anlegen, ändern, löschen)
- Externer Datenbestand – auch: Referenzdatenbestand– (ELF: External Logical File): fachliche Daten, die vom System nur gelesen werden

Definition

Eingabe: Hauptzweck: internen Datenbestand pflegen oder Systemverhalten ändern, wobei gilt:

- Daten oder Steuerinformationen kommen von außerhalb des Systems
- und mindestens ein ILF wird gepflegt, falls die Daten, die die Systemgrenze überqueren, keine Steuerinformationen sind, die das Systemverhalten verändern
- und mindestens eine der folgenden Bedingungen ist erfüllt (Einzigartigkeit):
  - Verarbeitungslogik ist einzigartig gegenüber anderen Eingaben
  - andere Datenfelder als bei anderen Eingaben werden verwendet
  - andere Datenbestände als bei anderen Eingaben werden verwendet

Unterscheidung der Funktionstypen auf Basis des Hauptzwecks eines Elementarprozesses.

Definition

Ausgabe:

- Hauptzweck: dem Anwender Informationen präsentieren
- Daten oder Steuerinformationen werden über Systemgrenze geschickt und
- Elementarprozess ist eindeutig (s.o.)
- und mindestens eine der folgenden Bedingungen ist erfüllt (Abgrenzung zu Abfrage):
  - Verarbeitungslogik enthält mindestens eine mathematische Formel oder Berechnung
  - Verarbeitungslogik pflegt mindestens einen ILF
  - Verarbeitungslogik verändert das Systemverhalten

Definition

Abfrage:

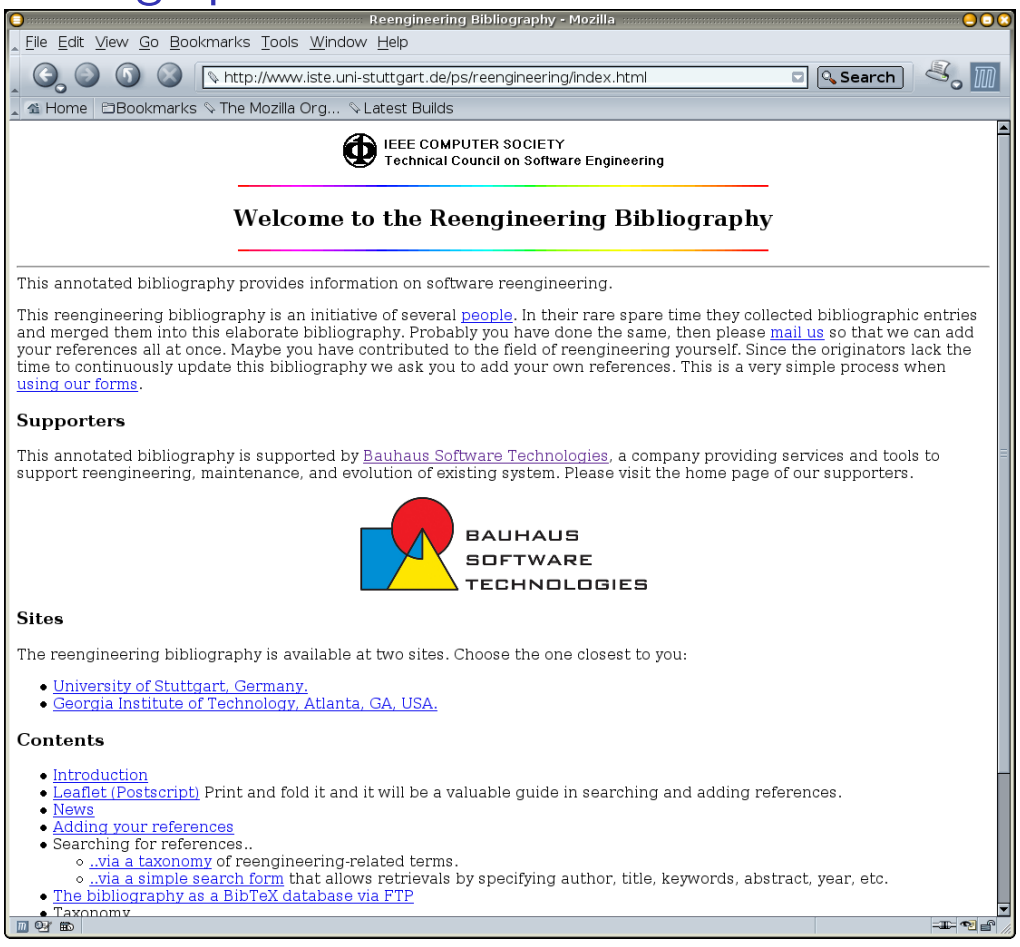
- Hauptzweck: dem Anwender Informationen präsentieren
- Daten oder Steuerinformationen werden über Systemgrenze geschickt und
- Elementarprozess ist eindeutig (s.o.) und
- und **alle** folgende Bedingungen sind erfüllt (Abgrenzung zu Ausgabe):
  - Elementarprozess bezieht Daten oder Steuerinformationen von einem ILF oder ELF
  - Verarbeitungslogik enthält keine mathematische Formel oder Berechnung
  - Verarbeitungslogik erzeugt keine abgeleiteten Daten
  - Verarbeitungslogik pflegt keinen ILF
  - Verarbeitungslogik verändert das Systemverhalten nicht

FP – Identifizieren von Funktionstypen

Schlüsselwörter geben Hinweise:

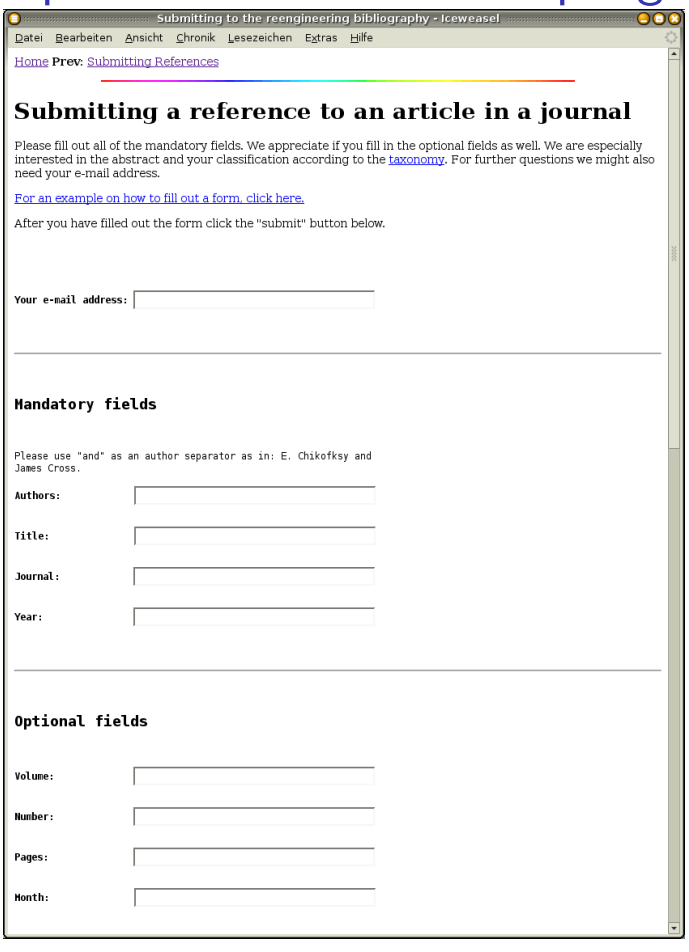
- EI: ablegen/speichern, de-/aktivieren, abbrechen, ändern/editieren/modifizieren/ersetzen, einfügen/hinzufügen, entfernen/löschen, erstellen, konvertieren, update, übertragen
- EO: anzeigen, ausgeben, ansehen, abfragen, suchen/durchsuchen, darstellen, drucken, selektieren, Anfrage, Abfrage, Report
- EQ: abfragen, anzeigen, auswählen, drucken, suchen/durchsuchen, darstellen/zeigen, drop down, extrahieren, finden, holen, selektieren, Ausgabe, Liste, Report

Online-Bibliographie: Startseite

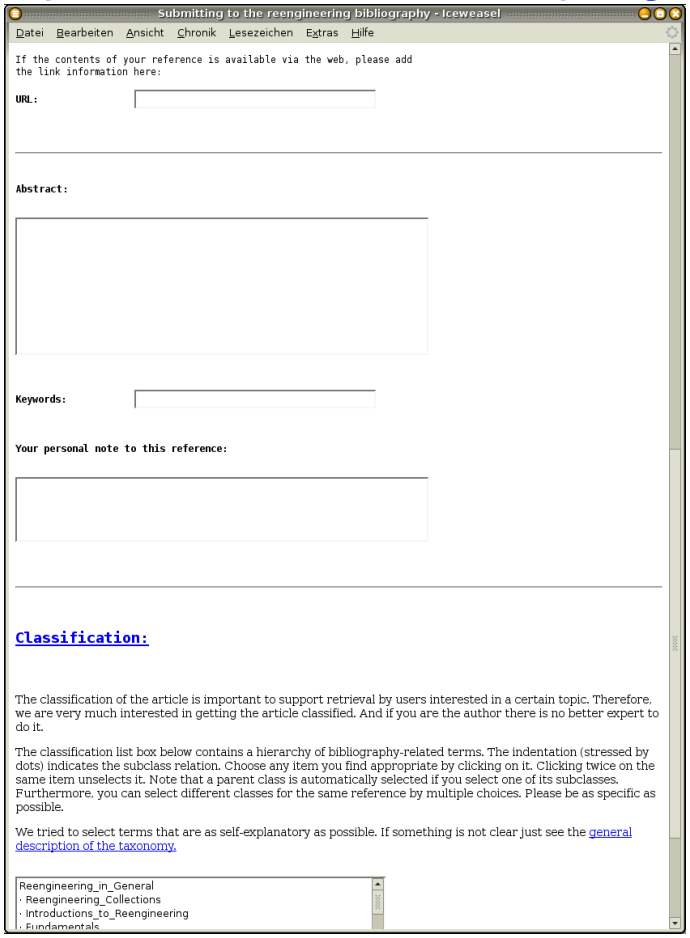

Reengineering Bibliography - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://www.iste.uni-stuttgart.de/ps/reengineering/index.html
Search
Home Bookmarks The Mozilla Org... Latest Builds
IEEE COMPUTER SOCIETY
Technical Council on Software Engineering
Welcome to the Reengineering Bibliography
This annotated bibliography provides information on software reengineering.
This reengineering bibliography is an initiative of several people. In their rare spare time they collected bibliographic entries and merged them into this elaborate bibliography. Probably you have done the same, then please mail us so that we can add your references all at once. Maybe you have contributed to the field of reengineering yourself. Since the originators lack the time to continuously update this bibliography we ask you to add your own references. This is a very simple process when using our forms.
Supporters
This annotated bibliography is supported by Bauhaus Software Technologies, a company providing services and tools to support reengineering, maintenance, and evolution of existing system. Please visit the home page of our supporters.
BAUHAUS SOFTWARE TECHNOLOGIES
Sites
The reengineering bibliography is available at two sites. Choose the one closest to you:
University of Stuttgart, Germany.
Georgia Institute of Technology, Atlanta, GA, USA.
Contents
Introduction
Leaflet (Postscript) Print and fold it and it will be a valuable guide in searching and adding references.
News
Adding your references
Searching for references..
..via a taxonomy of reengineering-related terms.
..via a simple search form that allows retrievals by specifying author, title, keywords, abstract, year, etc.
The bibliography as a BibTeX database via FTP
Taxonomy

Die Startseite enthält reine Navigationsinformationen. Diese Daten werden nicht gezählt. Weitere Elementarprozesse lassen sich aber auf dieser Seite identifizieren. Wir konzentrieren uns im Folgenden auf die besonders relevanten Elementarprozesse für das Hinzufügen von Referenzen, die Suche mittels Taxonomie und einem Suchformular.
Für den Benutzer von außen sichtbar bestehen die internen Datenbestände aus zwei logisch getrennten Inhalten. Zum einen sind das die Referenzen, die selbst in verschiedene Untergruppen zerfallen (Zeitschriftenartikel, Buch, Konferenzbeitrag etc.), zum anderen können wir die Taxonomie als Meta-Daten erkennen.
Externe Datenbestände, auf die das System zugreift, sind nicht zu erkennen.

Online-Bibliographie: Neuen Artikel einpflegen


Submitting to the reengineering bibliography - Iceweasel
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
Home Prev: Submitting References
Submitting a reference to an article in a journal
Please fill out all of the mandatory fields. We appreciate if you fill in the optional fields as well. We are especially interested in the abstract and your classification according to the taxonomy. For further questions we might also need your e-mail address.
For an example on how to fill out a form, click here.
After you have filled out the form click the "submit" button below.
Your e-mail address:
Mandatory fields
Please use "and" as an author separator as in: E. Chikofksy and James Cross.
Authors:
Title:
Journal:
Year:
Optional fields
Volume:
Number:
Pages:
Month:

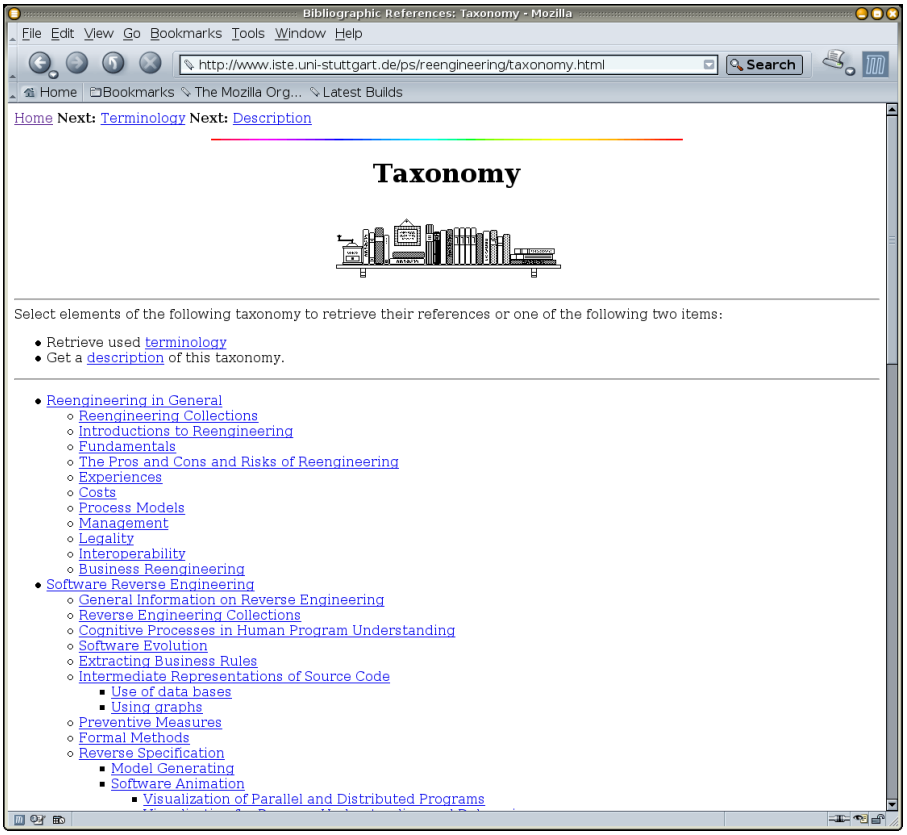
Online-Bibliographie: Neuen Artikel einpflegen


Submitting to the reengineering bibliography - Iceweasel
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
If the contents of your reference is available via the web, please add the link information here:
URL:
Abstract:
Keywords:
Your personal note to this reference:
Classification:
The classification of the article is important to support retrieval by users interested in a certain topic. Therefore, we are very much interested in getting the article classified. And if you are the author there is no better expert to do it.
The classification list box below contains a hierarchy of bibliography-related terms. The indentation (stressed by dots) indicates the subclass relation. Choose any item you find appropriate by clicking on it. Clicking twice on the same item unselects it. Note that a parent class is automatically selected if you select one of its subclasses. Furthermore, you can select different classes for the same reference by multiple choices. Please be as specific as possible.
We tried to select terms that are as self-explanatory as possible. If something is not clear just see the general description of the taxonomy.
Reengineering_in_General
· Reengineering_Collections
· Introductions_to_Reengineering
· Fundamentals

Mit dieser Seite werden neue Referenzen hinzugefügt. Dieser Elementarprozess hat als Hauptzweck die Eingabe. Die Taxonomie selbst wird nicht betroffen. Einzig der Datenbestand *Referenzen* wird verändert.
Über den Link *Classification* kann man sich die Taxnomie anzeigen lassen. Wir betrachten dies als eigenen Elementarprozess *Beschreibung der Taxonmie*. Er unterstützt zwar den anderen Elementarprozess, ist aber nicht Teil von diesem, da der andere Elementarprozess auch ohne ihn vollständig ist. Dieser Elementprozess ist eine klare Abfrage, da nur Daten ohne weitere Verarbeitung angezeigt werden. Ergebnis der Abfrage ist die Taxonomie als Liste (genauer: Baum) von Taxonomieeinträgen.

Insgesamt unterstützt diese Seite also zwei Elementarprozesse.

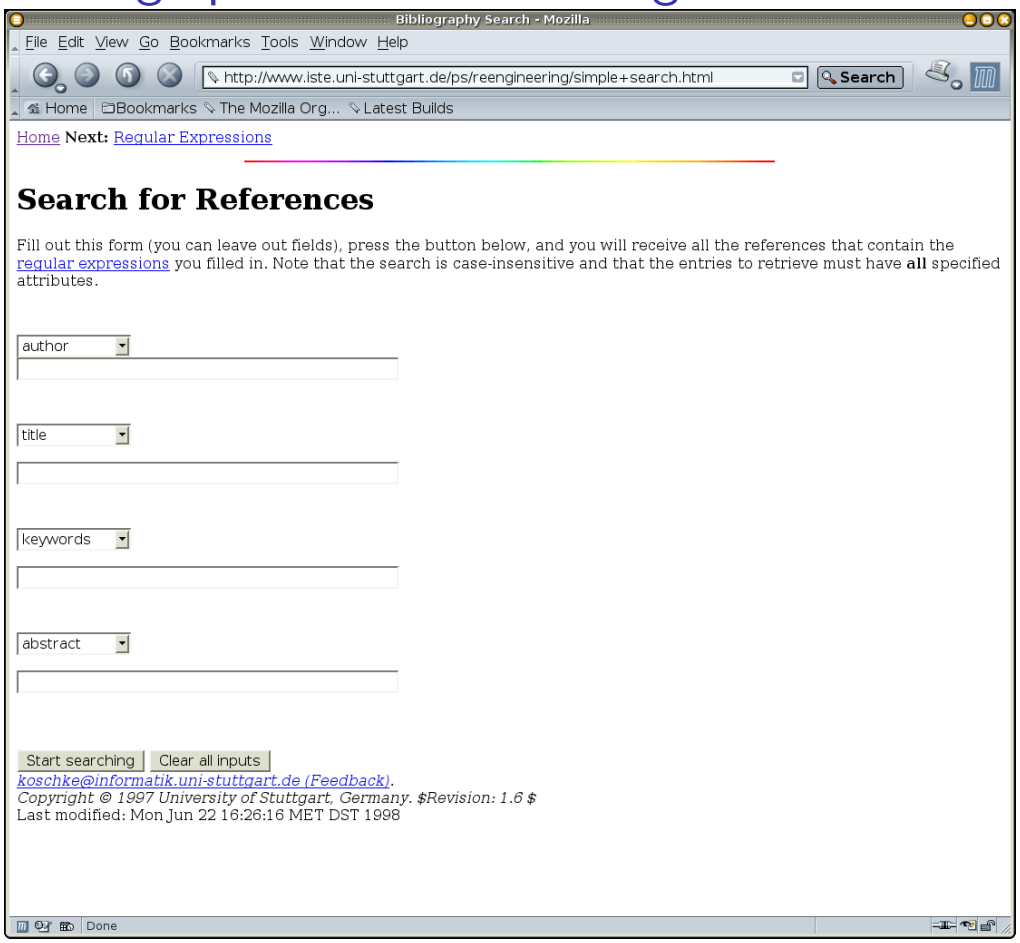
Online-Bibliographie: Taxonomiesuche



Annahme: Ein BibTeX-Eintrag habe max. 8 Einträge

Auf dieser Seite wird der Elementarprozess *Suche über die Taxonomie* erkennbar. Jeder Taxonomieeintrag stellt einen Link dar, der alle Artikel auflistet, die zu diesem Taxonomiepunkt gehören oder zu Taxonomiepunkten, die von diesem abgeleitet sind. Die Taxonomie selbst wird beim Start dieser Seite vom System bereit gestellt. Letzteres entspricht auf den ersten Blick dem Elementarprozess *Taxonomie anzeigen*. Wir müssen nun bestimmen, ob dieser Elementarprozess von dem weiter oben diskutierten unterscheidbar ist. Er unterscheidet sich nicht durch die Eingabe, die Verarbeitung oder den referenzierten Datenbeständen jedoch in den Ausgabedaten. Während beim ersten Elementarprozess *Beschreibung der Taxonomie* die einzelnen Taxonomiepunkte textuell beschrieben werden, wird uns auf dieser Seite beim Klicken auf einen Taxonomiepunkt angezeigt, welche Referenzen zu diesem Punkt existieren.
Ein weiteres Argument gegen das Vorliegen eines Elementarprozesses ist die Tatsache, dass dieser Elementarprozess weder eigenständig noch abgeschlossen ist. Er ist ganz offensichtlich integraler Bestandteil des Elementarprozesses *Suche über die Taxonomie*. Ohne die Anzeige der Taxonomie könnte der umfassende Elementarprozess nicht funktionieren.
Diese Seite unterstützt also nur einen Elementarprozess *Suche über die Taxonomie*. Es handelt sich bei diesem offensichtlich nicht primär um eine Eingabe, auch wenn der Benutzer einen Punkt der Taxonomie auswählt. Der Hauptzweck ist die Ausgabe von Daten. In Frage kommen somit zunächst Abfrage oder Ausgabe. Da die Verarbeitungsregel keine mathematische Formel oder Berechung erwarten lässt, keine Daten abgeleitet werden, keine ILF gepflegt wird und sich auch das Systemverhalten nicht ändern wird, handelt es sich klar um eine Abfrage.

Online-Bibliographie: Suche nach Eigenschaften


Bibliography Search - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://www.iste.uni-stuttgart.de/ps/reengineering/simple+search.html
Search
Home Bookmarks The Mozilla Org... Latest Builds
Home Next: Regular Expressions
Search for References
Fill out this form (you can leave out fields), press the button below, and you will receive all the references that contain the regular expressions you filled in. Note that the search is case-insensitive and that the entries to retrieve must have all specified attributes.
author
title
keywords
abstract
Start searching
Clear all inputs
koschke@informatik.uni-stuttgart.de (Feedback).
Copyright © 1997 University of Stuttgart, Germany. $Revision: 1.6 $
Last modified: Mon Jun 22 16:26:16 MET DST 1998
Done

Auf dieser Seite können Referenzen anhand von Stichwörtern gesucht werden. Die Suche nach Stichwörtern kann auf verschiedene Felder der Referenzen eingeschränkt werden. Den Suchbereich kann man mit Hilfe einer Listbox für jedes Stichwort auswählen.
Hier tritt die interessante und häufig gestellte Frage auf, wie Listboxen zu behandeln sind. Können Sie auch einen Elementarprozess darstellen? Poensgen und Bock (2005) geben hierauf die Antwort, dass eben nicht Listboxen, sondern Elementarprozesse bewertet werden. Ob mit einer Listbox ein Elementarprozess zu werten ist, hängt vom Einzelfall ab. Die Listbox hier repräsentiert zulässige BibTeX-Felder. Damit werden keine Daten aus einem fachlichen Datenbestand dargestellt, sondern Metadaten, die ein korrektes Attribut im internen Datenbestand *Referenzen* beschreiben. Die Function-Point-Methode verlangt, dass bei einer Abfrage oder Ausgabe Daten oder Steuerinformationen über die Anwendungsgrenze verschickt werden. Hierbei sind explizit fachliche Daten gemeint. Dies ist jedoch bei den Elementen der Listbox nicht der Fall.
Ein andere Situation läge vor, wenn die Listboxen fachliche Daten repräsentieren würden, beispielsweise, wenn aus einer Liste verzeichneter Autoren ausgesucht werden könnte.
Wir halten also fest, dass wir die Auswahl in den Listboxen nicht als Elementarprozess betrachten. Nichtsdestotrotz stellen sie Felder dar, die Eingaben darstellen. Sie werden mit übertragen, damit der Wert im Suchfeld richtig interpretiert werden kann. Somit sind sie zumindest für die Zählung der DETs relevant.

Beispiel: Online-Bibliographie

Elementarprozesse und Datenbestände der Online-Bibliographie:

- EI_1: Neuen Artikel einpflegen
- EQ_1: Beschreibung der Taxonomie
- EQ_2: Suche über Taxonomie
- EQ_3: Artikel über Suchmaske abfragen
- ILF_1: Referenzen
- ILF_2: Taxonomie-Metadaten

Function Points zusammenfassen

Unadjusted Function Points (UFP):

Parameter	Gewicht
EI	w_1
EO	w_2
EQ	w_3
ILF	w_4
ELF	w_5
UFP	$\sum w_i$

$\rightarrow$ zu ermitteln: w_i

Umfang $\sim$ Summe FPs; „ungewichtete Funktionspunkte“ (UFP)

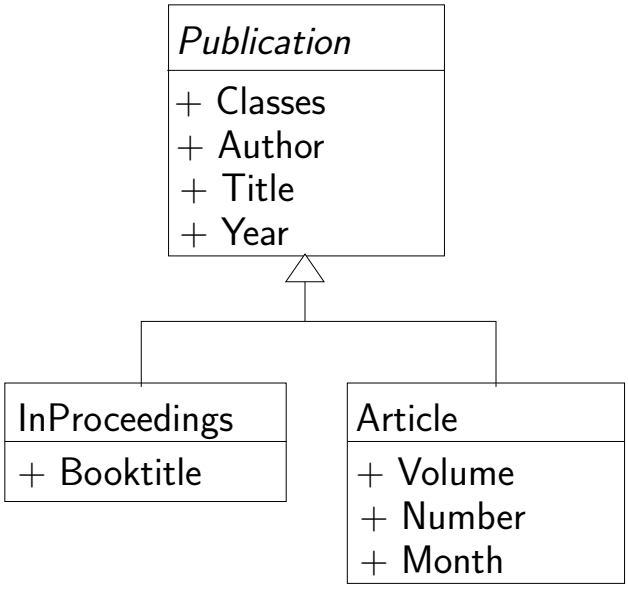
Beispiel: Komplexitätsgewichte w_i für ELF und ILF

Definition

Feldgruppe: RET (Record Element Type): für Benutzer erkennbare, logisch zusammengehörige Untergruppe von Datenelementen innerhalb eines Datenbestands (ILF, EIF)

Definition

Datenelementtypen: DET (Data Element Type): für Benutzer erkennbares, eindeutig bestimmbares, nicht-wiederholtes Feld



Hier werden die Daten abgeschätzt.
Der Datenbestand **Referenzen** unserer Online-Bibliographie-Beispiels wird hier beispielhaft (und fragmentarisch) durch eine Klassendiagramm beschrieben. Das angegebene Klassendiagramm hat insgesamt drei Klassen. Auf den ersten Blick scheinen hier drei Untergruppen zu existieren. Die Oberklasse ist jedoch abstrakt. Das heißt, dass eine solche Untergruppe gar nicht existiert bzw. nur eine logische Beschreibung darstellt, um Gemeinsamkeiten von anderen Untergruppen zusammen zu fassen. Abstrakte Klassen zählen somit nicht als Untergruppe. Die beiden anderen Klassen sind konkret. Somit ergeben sich zwei Feldgruppen: **RET = 2**. Die Datenelementtypen sind die Attribute aller Untergruppen einschließlich der von abstrakten Oberklassen ererbten. Davon gibt es acht: **DET = 8**.
Die **Taxonomie-Metadaten** sind wie folgt strukturiert. Wir haben einerseits den Namen des Taxonomiepunkts, dann einen beschreibenden Text und schließlich einen Verweis auf die Oberklasse. Damit ergeben sich **RET = 1 und DET = 3**.

FP – Bestimmung der Komplexitätsgewichte w_i

Komplexitätsmatrizen:

- (Funktionstyp, #FTRs/RETs, #DETs) $\rightarrow$ FPs
- Zählen mittels Zählregeln pro Funktionstyp

FTRs/RETs	DETs		
Funktionstyp	1 bis a	$a+1$ bis b	$> b$
1 bis x	gering	gering	mittel
$x+1$ bis y	gering	mittel	hoch
$> y$	mittel	hoch	hoch

FTR: number of files updated or referenced. A record element type is a user recognizable subgroup of data elements within an ILF or EIF. A data element type is a unique user recognizable, nonrecursive, field.
Zählen von Datenelementtypen (DET)
Ein Datenelementtyp (DET) ist aus der Benutzersicht eindeutig bestimmbares, nicht rekursives Feld, das von der zu bewertenden externen Eingabe (EI) in einem internen Datenbestand (ILF) gepflegt wird.
Zählen Sie 1 DET für jedes aus Benutzersicht nicht rekursive Feld, das die Systemgrenze kreuzt und gebraucht wird, um den Elementarprozess abzuschließen.
Beispiel: Ein Texteingabefeld, in dem der Nachname eines neuen Kunden eingegeben wird, wird als 1 DET gezählt.
Gegenbeispiel: Eine Dateiauswahlliste, in der beliebig viele Dateien von der Festplatte des Benutzers ausgewählt werden können, ist rekursiv, und muss somit gesondert gezählt werden.
Zählen Sie keine Felder, die durch das System gesucht und/oder in einem ILF gespeichert werden, wenn die Daten nicht die Systemgrenze überqueren.
Zählen Sie nur 1 DET für die Fähigkeit, eine Systemantwort als Meldung für einen aufgetretenen Fehler aus der Anwendung heraus zu senden bzw. für die Bestätigung, dass die Verarbeitung beendet oder fortgesetzt werden kann Beispiel: Bei Eingabe eines Datums soll z.B. das Format TT/MM/JJJJ eingehalten werden. Gibt der Bearbeiter z.B. '12.03.1997' ein und bestätigt seine Eingabe, so erhält er die Meldung 'neuer Datensatz gespeichert'. Gibt er hingegen '12.3.97' ein (Jahreszahl nicht vierstellig) so erhält er die Fehlermeldung 'Fehler: Bitte Datum korrigieren'. Nur ein DET wird für diese Fähigkeit des Systems gezählt.

Zählen Sie nur 1 DET für die Möglichkeit, eine Aktion durchzuführen, auch wenn es viele Methoden gibt, die denselben logischen Prozess anstoßen. Beispiel: In einem Eingabeformular gibt es einen OK-Button zum Absenden der Daten. Die Tastaturkombination STRG-S führt ebenfalls zum Senden der Daten. Somit wird nur ein DET gezählt.

(Fortsetzung)

Zählen von referenzierte Datenbeständen (FTR)

Ein referenzierter Datenbestand (FTR) ist eine vom Benutzer definierte Gruppierung zusammengehöriger Daten oder Steuerungsinformationen in einem internen Datenbestand (ILF), die bei der Bearbeitung der externen Eingabe gelesen oder gepflegt wird.

Zählen Sie 1 FTR für jeden referenzierten Datenbestand, der während der Verarbeitung der externen Eingabe gelesen wird. Beispiel: Es werden durch eine externe Eingabe Produktdaten in einer Datenbank gespeichert. Dazu werden die Produktbezeichnungen aus einer weiteren Datenbank ausgelesen, die damit zusätzlich zu der zu aktualisierenden Produktdatenbank einen weiteren Datenbestand darstellt, der jedoch nur gelesen wird.

Zählen Sie 1 FTR für jede ILF, die während der Verarbeitung der externen Eingabe gepflegt wird. Beispiel: Es wird zusätzlich zu den Aktionen des vorigen Beispiels eine Textdatei aktualisiert, in der die Anzahl der Zugriffe auf die Datenbank verzeichnet wird.

Zählen Sie nur 1 FTR für jede ILF, die während der Verarbeitung der externen Eingabe gelesen und gepflegt wird.

Beispiel: Würden die Informationen der Textdatei ebenfalls in der Datenbank gespeichert werden, so wird diese nur

als 1 FTR gezählt, obwohl die Datenbank zur Ein- und Ausgabe von Daten verwendet wird.

(Fortsetzung)

Besonderheiten bei grafischen Benutzungsoberflächen

Optionsfelder (Radiobuttons) stellen Datenelemente dar. Es wird pro Gruppe von Optionsfeldern 1 DET gezählt, da innerhalb einer Gruppe nur ein Optionsfeld ausgewählt werden kann. Beispiel: Eine Gruppe von 12 Radiobuttons, in der ein PKW-Typ ausgewählt werden kann, wird als 1 DET gezählt.

Kontrollkästchen (Checkboxen) stellen Datenelemente dar. Im Gegensatz zu Optionsfeldern können aus einer Gruppe von Checkboxen mehrere Elemente gleichzeitig ausgewählt werden. Somit wird jedes Element als 1 DET gezählt. Beispiel: Eine Gruppe von 12 Checkboxen, mit der ein Pizza-Belag zusammengestellt werden kann, wird als 12 DETs gezählt.

Eingabe- und Ausgabefelder stellen Datenelemente dar. Beispiel: In einer Bildschirmmaske werden Vorname, Nachname, Straße, Hausnummer, PLZ und Ort in Eingabefeldern erfasst. Somit werden 6 DET gezählt.

Literale stellen keine Datenelemente dar. Beispiel: Vor einem Feld ist die Textzeile 'monatliches Gehalt' und dahinter 'in Euro/Monat' angegeben. Beide Textzeilen sind Literale und werden nicht gezählt

Enter-, OK-Button und Programmfunktionstaste werden insgesamt als 1 DET gezählt, da jeweils die gleiche Funktion ausgeführt wird. Beispiel: Die Daten eines Dialogs werden nach Betätigen der Enter-Taste oder nach Betätigen der Schaltfläche 'übernehmen' (gleiche Funktion wie OK-Button) in einer Datenbank gespeichert. Es wird 1 DET gezählt.

Berichte/Reports können verschiedene Ausgabeformen haben. So kann die gleiche Datenbasis zur Darstellung als Tortendiagramm, Tabelle, textuell, als druckbares Format oder als Exportdatei dargestellt werden. Jedes Format stellt dabei eine externe Ausgabe (EO) dar.

FP – Werte der Komplexitätsgewichte w_i

RETs \ DETs	**ILF**	1-19	20-50	51+
	1	7	7	10
	2-5	7	10	15
	6+	10	15	15

RETs \ DETs	**ELF**	1-19	20-50	51+
	1	5	5	7
	2-5	5	7	10
	6+	7	10	10

Function Points zusammenfassen

Parameter	RET	DET	Gewicht
ILF_1	2	8	7
ILF_2	1	3	7
Parameter	FTR	DET	Gewicht
EI_1			
EQ_1			
EQ_2			
EQ_3			
UFP			

Komplexitätsgewichte w_i für EI, EO, EQ

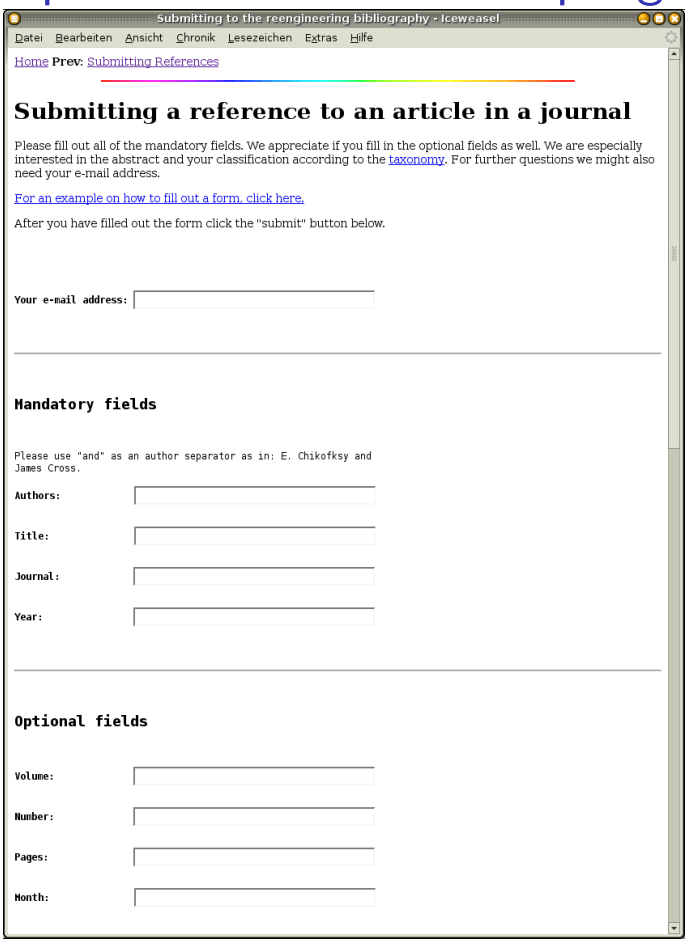
Definition

Referenzierte Datenbestände: FTR (File Type Referenced):
von Elementarprozess verwendeter Datenbestand (ILF, ELF)

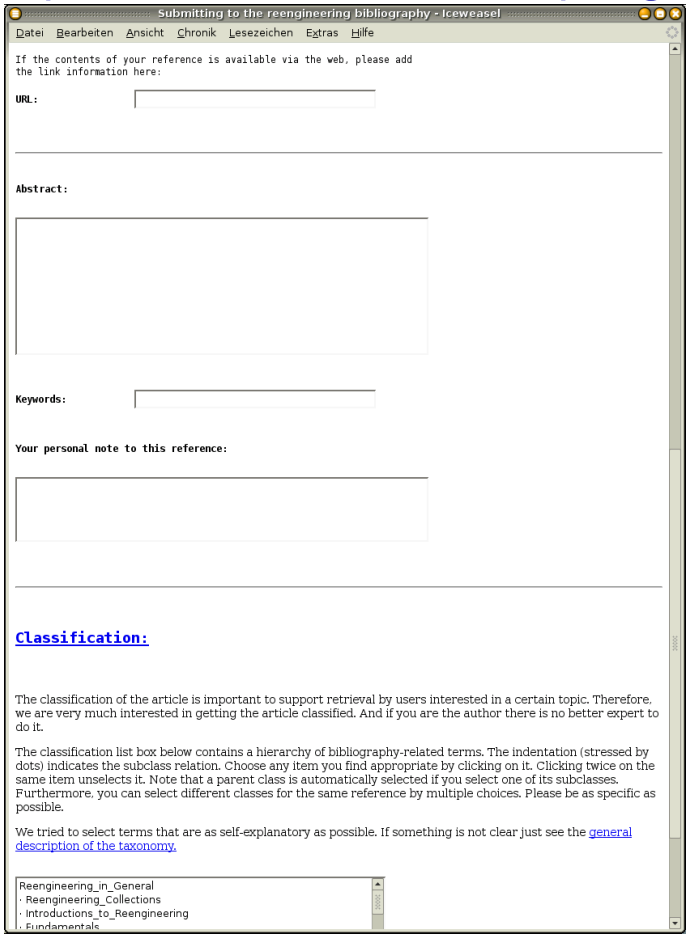
Beispiel: der Kundenstammdatenbestand, der bei der Ausgabe von Kundendaten herangezogen wird

Beispiele für DETs im Kontext von Funktionen:
Eingabe-/Ausgabefelder (GUI), Spalten u.Ä. bei Berichten

Online-Bibliographie: Neuen Artikel einpflegen


Submitting to the reengineering bibliography - Iceweasel
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
Home Prev: Submitting References
Submitting a reference to an article in a journal
Please fill out all of the mandatory fields. We appreciate if you fill in the optional fields as well. We are especially interested in the abstract and your classification according to the taxonomy. For further questions we might also need your e-mail address.
For an example on how to fill out a form, click here.
After you have filled out the form click the "submit" button below.
Your e-mail address:
Mandatory fields
Please use "and" as an author separator as in: E. Chikofksy and James Cross.
Authors:
Title:
Journal:
Year:
Optional fields
Volume:
Number:
Pages:
Month:

Online-Bibliographie: Neuen Artikel einpflegen


Submitting to the reengineering bibliography - Iceweasel
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
If the contents of your reference is available via the web, please add the link information here:
URL:
Abstract:
Keywords:
Your personal note to this reference:
Classification:
The classification of the article is important to support retrieval by users interested in a certain topic. Therefore, we are very much interested in getting the article classified. And if you are the author there is no better expert to do it.
The classification list box below contains a hierarchy of bibliography-related terms. The indentation (stressed by dots) indicates the subclass relation. Choose any item you find appropriate by clicking on it. Clicking twice on the same item unselects it. Note that a parent class is automatically selected if you select one of its subclasses. Furthermore, you can select different classes for the same reference by multiple choices. Please be as specific as possible.
We tried to select terms that are as self-explanatory as possible. If something is not clear just see the general description of the taxonomy.
Reengineering_in_General
· Reengineering_Collections
· Introductions_to_Reengineering
· Fundamentals

Beim Elementarprozesse *Neuen Artikel einpflegen* EI_1 sind insgesamt 13 Textfelder auszufüllen. Außerdem muss am Ende der Schalter für das Absenden der Daten betätigt werden, um den Elementarprozess abzuschließen (ein Schalter für das Abbrechen zählt nicht, da damit der Prozess nicht abgeschlossen wird). Für diesen Elementarprozess ergeben sich also zunächst 14 DETs.
Außerdem soll der Artikel in der Taxonomie eingeordnet werden. Dies zählt als weitere Eingabemöglichkeit. Dabei können mehrere Klassen der Taxonomie ausgewählt werden. Der Datentyp der Auswahl, die dem System übergeben wird, stellt somit eine Liste von Taxonomieeinträgen dar. Die Taxonomieeinordnung zählt somit als ein 1 DET. Die Eingabe EI_1 hat somit 15 DETs
Die Taxonomieeinordnung stellt keinen Elementarprozess dar, weil damit kein abgeschlossener Benutzerzweck verbunden ist. Sie ist nur sinnvoll im Kontext des Elementarprozesses *Neuen Artikel einpflegen*.
Wenn der Link *Classification* betätigt wird, wird die Taxonomie als Baum von Taxonomieeinträgen angezeigt. Dies zählt als ein DET (nicht die Anzahl von Daten sondern der Datentyp wird gezählt). Der Link *Classification* ist lediglich eine Navigation und zählt somit nicht. Die Abfrage EQ_1 hat somit nur einen DET.
Nun müssen noch die FTRs der beiden Elementarprozesse bestimmt werden. Bei der Eingabe eines neuen Artikels muss nur der Datenbestand *Referenzen* angefasst werden, bei der Abfrage der Taxonomie nur der Datenbestand *Taxonomie*.

Function Points zusammenfassen

- EI_1: Neuen Artikel einpflegen
- EQ_1: Beschreibung der Taxonomie
- EQ_2: Suche über Taxonomie
- EQ_3: Artikel über Suchmaske abfragen
- ILF_1: Referenzen
- ILF_2: Taxonomie-Metadaten

Parameter	RET	DET	Gewicht
ILF_1	2	8	7
ILF_2	1	3	7
Parameter	FTR	DET	Gewicht
EI_1	1	15	
EQ_1	1	1	
EQ_2			
EQ_3			
UFP			

FP – Werte der Komplexitätsgewichte w_i

FTRs \ DETs	**EI**	1-4	5-15	16+
	0-1	3	3	4
	2	3	4	6
	3+	4	6	6

FTRs \ DETs	**EO**	1-5	6-19	20+
	0-1	4	4	5
	2-3	4	5	7
	4+	5	7	7

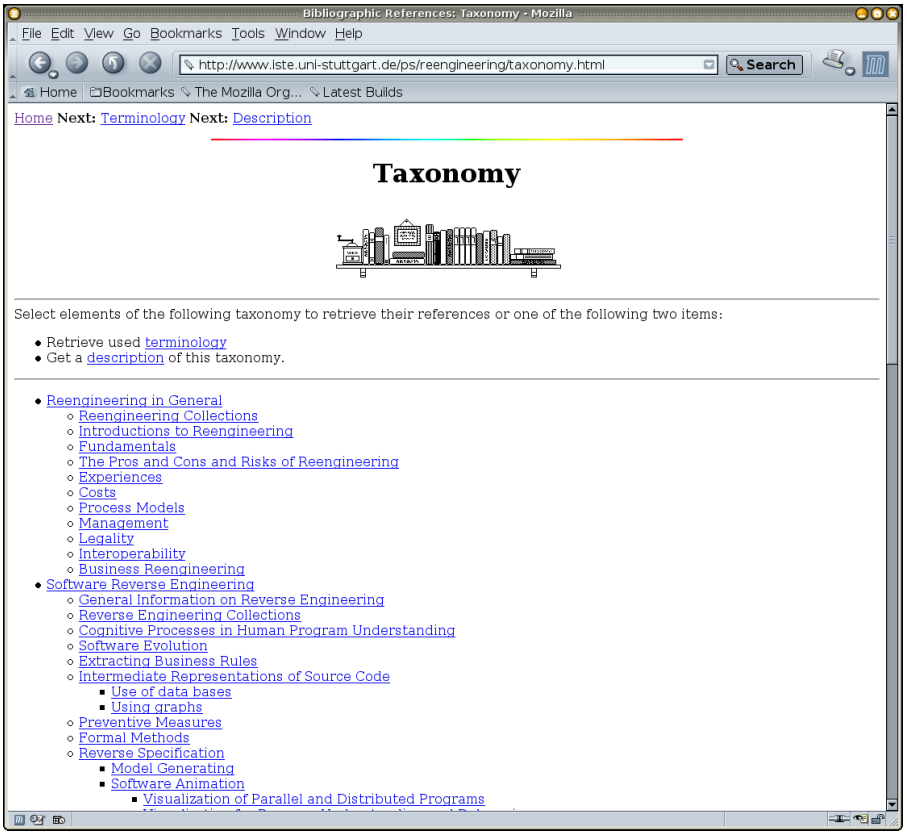
FTRs \ DETs	**EQ**	1-5	6-19	20+
	0-1	3	3	4
	2-3	3	4	6
	4+	4	6	6

Function Points zusammenfassen

- EI_1: Neuen Artikel einpflegen
- EQ_1: Beschreibung der Taxonomie
- EQ_2: Suche über Taxonomie
- EQ_3: Artikel über Suchmaske abfragen
- ILF_1: Referenzen
- ILF_2: Taxonomie-Metadaten

Parameter	RET	DET	Gewicht
ILF_1	2	8	7
ILF_2	1	3	7
Parameter	FTR	DET	Gewicht
EI_1	1	15	3
EQ_1	1	1	3
EQ_2			
EQ_3			
UFP			

Online-Bibliographie: Taxonomiesuche



Annahme: Ein BibTeX-Eintrag habe max. 8 Einträge

Auf dieser Seite wird der Elementarprozess *Suche über die Taxonomie* erkennbar. Jeder Taxonomieeintrag stellt einen Link dar, der alle Artikel auflistet, die zu diesem Taxonomiepunkt gehören oder zu Taxonomiepunkten, die von diesem abgeleitet sind. Die Taxonomie selbst wird beim Start dieser Seite vom System bereit gestellt. Letzteres entspricht auf den ersten Blick dem Elementarprozess *Taxonomie anzeigen*. Wir müssen nun bestimmen, ob dieser Elementarprozess von dem weiter oben diskutierten unterscheidbar ist. Er unterscheidet sich nicht durch die Eingabe, die Verarbeitung oder den referenzierten Datenbeständen jedoch in den Ausgabedaten. Während beim ersten Elementarprozess *Beschreibung der Taxonomie* die einzelnen Taxonomiepunkte textuell beschrieben werden, wird uns auf dieser Seite beim Klicken auf einen Taxonomiepunkt angezeigt, welche Referenzen zu diesem Punkt existieren.
Ein weiteres Argument gegen das Vorliegen eines Elementarprozesses ist die Tatsache, dass dieser Elementarprozess weder eigenständig noch abgeschlossen ist. Er ist ganz offensichtlich integraler Bestandteil des Elementarprozesses *Suche über die Taxonomie*. Ohne die Anzeige der Taxonomie könnte der umfassende Elementarprozess nicht funktionieren.
Diese Seite unterstützt also nur einen Elementarprozess *Suche über die Taxonomie*. Es handelt sich bei diesem offensichtlich nicht primär um eine Eingabe, auch wenn der Benutzer einen Punkt der Taxonomie auswählt. Der Hauptzweck ist die Ausgabe von Daten. In Frage kommen somit zunächst Abfrage oder Ausgabe. Da die Verarbeitungsregel keine mathematische Formel oder Berechung erwarten lässt, keine Daten abgeleitet werden, keine ILF gepflegt wird und sich auch das Systemverhalten nicht ändern wird, handelt es sich klar um eine Abfrage.

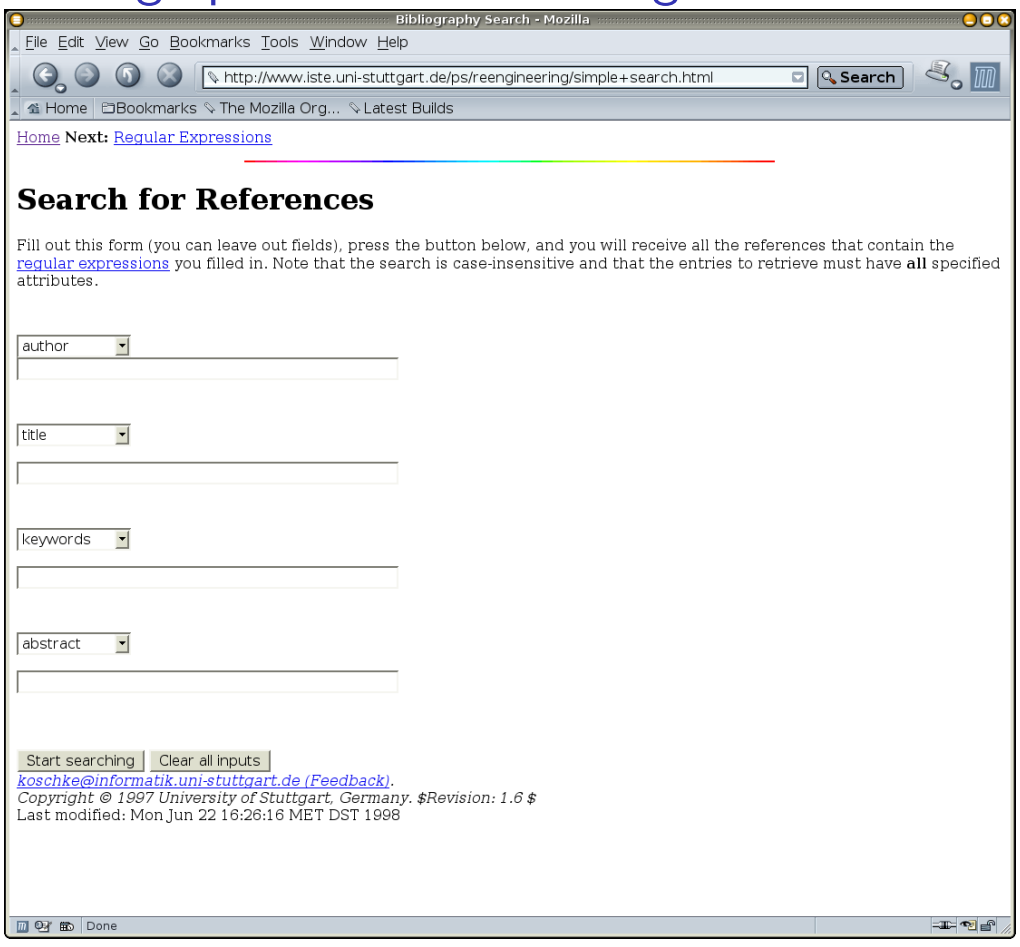
Zu beachten ist bei der Zählung der DETs generell, dass jedes DET nur einmal gezählt wird, auch wenn es in beide Richtungen der Systemgrenze übertragen wird.
Der Elementarprozess *Suche über die Taxonomie* EQ_2 ist mit einem DET für den ausgewählten Taxonomiepunkt assoziiert. Das Resultat ist eine Liste bibliographischer Referenzen, die der BibTeX-Struktur folgen. Da im Prinzip jede Art von Referenz zurückgeliefert werden kann, müssen wir die maximale Anzahl von Feldern aller BibTeX-Referenztypen bestimmen. Ohne in die Untiefen von BibTeX einzutauchen, nehmen wir der Einfachheit halber an, wir hätten maximal 8 Felder. Damit ergeben sich dann insgesamt 8+1=9 DETs für diesen Elementarprozess.
Der Elementprozess muss sowohl den Taxonomie- als auch den Referenzendatenbestand betrachten. Damit ergeben sich zwei FTRs.

Function Points zusammenfassen

- EI_1: Neuen Artikel einpflegen
- EQ_1: Beschreibung der Taxonomie
- EQ_2: Suche über Taxonomie
- EQ_3: Artikel über Suchmaske abfragen
- ILF_1: Referenzen
- ILF_2: Taxonomie-Metadaten

Parameter	RET	DET	Gewicht
ILF_1	2	8	7
ILF_2	1	3	7
Parameter	FTR	DET	Gewicht
EI_1	1	15	3
EQ_1	1	1	3
EQ_2	2	9	4
EQ_3			
UFP			

Online-Bibliographie: Suche nach Eigenschaften


Bibliography Search - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://www.iste.uni-stuttgart.de/ps/reengineering/simple+search.html
Search
Home
Bookmarks
The Mozilla Org...
Latest Builds
Home Next: Regular Expressions
Search for References
Fill out this form (you can leave out fields), press the button below, and you will receive all the references that contain the regular expressions you filled in. Note that the search is case-insensitive and that the entries to retrieve must have all specified attributes.
author
title
keywords
abstract
Start searching
Clear all inputs
koschke@informatik.uni-stuttgart.de (Feedback).
Copyright © 1997 University of Stuttgart, Germany. $Revision: 1.6 $
Last modified: Mon Jun 22 16:26:16 MET DST 1998
Done

Auf dieser Seite können Referenzen anhand von Stichwörtern gesucht werden. Die Suche nach Stichwörtern kann auf verschiedene Felder der Referenzen eingeschränkt werden. Den Suchbereich kann man mit Hilfe einer Listbox für jedes Stichwort auswählen.
Hier tritt die interessante und häufig gestellte Frage auf, wie Listboxen zu behandeln sind. Können Sie auch einen Elementarprozess darstellen? Poensgen und Bock (2005) geben hierauf die Antwort, dass eben nicht Listboxen, sondern Elementarprozesse bewertet werden. Ob mit einer Listbox ein Elementarprozess zu werten ist, hängt vom Einzelfall ab. Die Listbox hier repräsentiert zulässige BibTeX-Felder. Damit werden keine Daten aus einem fachlichen Datenbestand dargestellt, sondern Metadaten, die ein korrektes Attribut im internen Datenbestand *Referenzen* beschreiben. Die Function-Point-Methode verlangt, dass bei einer Abfrage oder Ausgabe Daten oder Steuerinformationen über die Anwendungsgrenze verschickt werden. Hierbei sind explizit fachliche Daten gemeint. Dies ist jedoch bei den Elementen der Listbox nicht der Fall.
Ein andere Situation läge vor, wenn die Listboxen fachliche Daten repräsentieren würden, beispielsweise, wenn aus einer Liste verzeichneter Autoren ausgesucht werden könnte.
Wir halten also fest, dass wir die Auswahl in den Listboxen nicht als Elementarprozess betrachten. Nichtsdestotrotz stellen sie Felder dar, die Eingaben darstellen. Sie werden mit übertragen, damit der Wert im Suchfeld richtig interpretiert werden kann. Somit sind sie zumindest für die Zählung der DETs relevant.

Insgesamt ergeben sich seitens der Eingabe durch den Benutzer für den Elementarprozess EQ_3 vier DETs für die Textfelder, weitere vier DETs für die Listboxen und schließlich noch ein DET für den Schalter, um die Suche zu starten, der den Elementarprozess anstößt.
Als Resultat erhalten wir eine Liste aller Referenzen, die die angegebenen Suchkriterien erfüllen. Da auch hier wieder die Referenztypen sehr unterschiedlich sein können, gehen wir wie bereits weiter oben von einer Obergrenze von 8 verschiedenen BibTeX-Attributen aus.
Die Gesamtzahl der DETs für die Eingabe und Ausgabe dieses Elementarprozesses beträgt somit $9+8=17$.
Die Suche ist unabhängig von der Taxonomie, so dass nur der Datenbestand *Referenzen* angefasst werden muss. Es ergibt sich damit ein FTR.

Function Points zusammenfassen

- EI_1: Neuen Artikel einpflegen
- EQ_1: Beschreibung der Taxonomie
- EQ_2: Suche über Taxonomie
- EQ_3: Artikel über Suchmaske abfragen
- ILF_1: Referenzen
- ILF_2: Taxonomie-Metadaten

Parameter	RET	DET	Gewicht
ILF_1	2	8	7
ILF_2	1	3	7
Parameter	FTR	DET	Gewicht
EI_1	1	15	3
EQ_1	1	1	3
EQ_2	2	9	4
EQ_3	1	17	4
UFP			28

FP – Gewichtete Function-Points

Systemmerkmale:

- Datenkommunikation
- Verteilte Verarbeitung
- Leistungsanforderungen
- Ressourcennutzung
- Transaktionsrate
- Online-Benutzerschnittstelle
- Benutzerfreundlichkeit
- Online-Verarbeitung
- Komplexe Verarbeitung
- Wiederverwendbarkeit
- Migrations-/Installationshilfen
- Betriebshilfen
- Mehrfachinstallationen
- Änderungsfreundlichkeit

Bewertung: 0 = kein Einfluss, 5 = starker Einfluss

Konkrete Fragen I

- Are data communications required? **1**
- Are there distributed processing functions? **0**
- Is performance critical? **1**
- Will the system run in an existing, heavily utilized operational environment? **1**
- How frequently are transactions executed? **3**
- Does the system require on-line data entry? **4**
- Was the application designed for end-user efficiency? **0**

Die Zahlen beziehen sich auf unser laufendes Beispiel.

Konkrete Fragen II

- Are the master files updated on-line? **0**
- Is the internal processing complex? **1**
- Is the code designed to be reusable? **1**
- Are conversion and installation included in the design? **0**
- How effective and/or automated are start-up, back-up, and recovery procedures? **0**
- Is the system designed for multiple installations in different organizations? **2**
- Is the application designed to facilitate change and ease of use by the user? **0**

FP – Gewichtete Function-Points

- TDI (Total Degree of Influence) = Summe der Bewertungen $\in [0 \ldots 14 * 5 = 70]$
- VAF (Value Adjustment Factor) = TDI/100 + 0,65
  $\rightarrow$ Gesamteinflussfaktor: 65% - 135%
- AFP (Adjusted Function Points) = UFP $\cdot$ VAF

Beispiel:

- TDI = 14
- VAF = 14/100 + 0,65 = 0,79
- AFP = 28 $\cdot$ 0,79 = 23 (es wird stets aufgerundet)

Kritik an der Function-Point-Methode

Einwand gegen Systemmerkmale:

- einige der Systemmerkmale sind heute eher anachronistisch
- durch Multiplikation von VAF und UAF findet eine fragwürdige Vermengung von technischen Faktoren und funktionalem Umfang sowie von quantitativen und qualitativen Aspekten statt
- VAF bringt kaum praktischen Nutzen:
  - COCOMO verwendet UAF als Eingangsgröße
  - COCOMO hat eigene technische und organisatorische Gewichtsfaktoren

→ AFP sind heute eher unbedeutend (im Gegensatz zu UFP)

→ bei Angaben von Function-Points nachfragen: AFP oder UFP?

Kritik an der Function-Point-Methode

Allgemeinere Einwände:

- sehr stark auf Informationssysteme bezogen; Übertragbarkeit auf andere Arten, wie z.B. reaktive Systeme oder Compiler, fragwürdig
- die Komplexität der Anfragen kann maximal um den Faktor 2 variieren; die Komplexität der Verarbeitungslogik geht nicht direkt ein

FP – Umrechnung in Aufwand

Gesucht: Abbildung FPs $\rightarrow$ Aufwand

- Erstellung einer neuen Erfahrungskurve
  (Zählen abgeschlossener Projekte, Regressionsanalyse)
- Datenbank, z.B. ISBSG[2]:
  - 3GL-Projekte: $\text{PM} = 0,971 \cdot \text{AFP}^{0,351}$
  - 4GL-Projekte: $\text{PM} = 0,622 \cdot \text{AFP}^{0,405}$
  - basierend auf 662 Projekten: $\text{PM} = 0,38 \cdot \text{AFP}^{0,37}$
- grobe Schätzung mit Faustregeln Jones (1996):
  - Entwicklungsdauer (Monate) $= AFP^{0,4}$
  - Personen = AFP / 150 (aufgerundet)
  - Aufwand (Personenmonate) = Personen $\cdot$ Entwicklungsdauer

Beispiel (mit Jones-Schätzung):

- Entwicklungsdauer (Monate) $= 23^{0,4} = 3,5$
- Personen $= 23/150 \rightarrow 1$
- Aufwand (Personenmonate) $= 1 \cdot 3{,}5 = 3{,}5$

[2] International Software Benchmarking Standards Group

http://www.isbsg.org/ http://www.isbsg.org/isbsg.nsf/weben/Project%20Duration

FP – Umrechnung in LOC

Mittlere Anzahl Codezeilen pro FP (Jones 1995):

Sprache	ØLOC
Assembler	320
C	128
FORTRAN	107
COBOL (ANSI 85)	91
Pascal	91
C++	53
Java	53
Ada 95	49
Smalltalk	21
SQL	12

Bewertung der Function-Point-Methode

+ Wird als beste verfügbare Methode zur Schätzung kommerzieller Anwendungssysteme angesehen (Balzert 1997)
+ Sinnvoll einsetzbar, wenn Erfahrungswerte von vergleichbaren Projekten vorliegen (Kemerer 1987)
– Bewertung der Systemmerkmale subjektiv (Symons 1988)
+ Studie: mittlere FP-Abweichung zwischen zwei „Zählern“ nur 12% (Kemerer und Porter 1992)
– Zählen der FPs relativ aufwendig

Object Points

- Für 4GLs (Query Languages, Report Writers, . . .)
- Haben nicht unbedingt mit OOP-Objekten zu tun
- Gewichtete Schätzung von *Objects Points* =
  - Anzahl verschiedener „Screens“
  - Anzahl erstellter „Reports“
  - Anzahl zu entwickelnder 3GL-Module
- Vorteil: Einfacher und weniger zu schätzen:
  vergleichbare Präzision wie Function-Point-Schätzung (Banker u. a. 1991)
  47% des Aufwands für Function-Point-Schätzung (Kauffman und Kumar 1993)

Object Points

Screens

# views contained	# data tables		
	< 4	< 8	8+
< 3	1	1	2
3-7	1	2	3
≥ 8	2	3	3

Reports

# sections contained	# data tables		
	< 4	< 8	8+
0-1	2	2	5
2-3	2	5	8
4+	5	8	8

Views: Menge logisch zusammengehöriger Daten (z.B. Kundenstammdaten)

Data Tables = # Server data tables + # Client data tables (Tabellen, die abgefragt werden müssen, um Daten zu bestimmen)

Jede 3GL-Komponente: 10 object points

COCOMO

COCOMO = Constructive Cost Model (Boehm 1981)

- Eingaben: Systemgröße, Projektrahmenbedingungen
- Ausgaben: Realisierungsaufwand, Entwicklungszeit
- Basiert auf Auswertung sehr vieler Projekte

COCOMO II

Unterscheidung nach Phasen (Boehm u. a. 2000):

- Frühe Prototypenstufe
- Frühe Entwurfsstufe
- Stufe nach Architekturentwurf

Spätere Schätzung $\rightarrow$ höhere Genauigkeit

COCOMO II – Early prototyping level

Eingaben:

- Object Points (OP)
- Produktivität (PROD):

Erfahrung/Fähigkeiten der Entwickler	- -	-	o	+	++
Reife/Fähigkeiten der CASE-Tools	- -	-	o	+	++
Median obiger Zeilen	- -	-	o	+	++
PROD [NOP/Monat]	4	7	13	25	50

- Wiederverwendungsanteil *%reuse* in Prozent

Abgeleitete Größen:

- New Object Points (NOP): berücksichtigen Wiederverwendung
  $NOP = \text{OP} \cdot (100 - \%\text{reuse})/100$
- Aufwand in Personenmonaten $\text{PM} = \text{NOP}/\text{PROD}$

COCOMO II – Early design level

- Eingabe: FP $\rightarrow$ LOC
- Ausgabe: Personenmonate $PM_{NS} = A \cdot \text{KLOC}^{E} \cdot EM + \text{PM}_{\text{m}}$ bei nominalem Zeitplan

$$E = B + \sum w_i/100$$

w_i: 5 = sehr klein, 0 = sehr groß:

- Erfahrung mit Domäne
- Flexibilität des Entwicklungsprozesses
- Risikomanagement
- Teamzusammenhalt
- Prozessreife

Schätzung basiert auf Function Points (LOCs werden daraus abgeleitet).$A = 2,94$ in initialer Kalibrierung (empirischer Wert)Nach Kalibrierung für COCOMO II.2000 $B = 0,91$; ein Wert > 1 wäre plausibler.Man beachte, dass alle Gewichte w_i, die in den Exponenten E einfließen, nichts mit der Technik sondern nur mit organisatorischen Dingen (Erfahrung, Prozess, Team) zu tun haben.

COCOMO II – Early design level

$$PM_{NS} = A \cdot \mathrm{KLOC}^{E} \cdot EM + \mathrm{PM_m}$$

$$EM = \prod \mathit{Effort\text{-}Multiplier}_{\mathrm{i}}$$

7 lineare Einflussfaktoren (6 Stufen, Standard ist 1,0; in Tabelle nachzuschlagen)

- Produktgüte
- Produktkomplexität
- Plattformkomplexität
- Fähigkeiten des Personals
- Erfahrung des Personals
- Zeitplan
- Infrastruktur

Hier kommen nun technische Aspekte und Anforderungen an die Software mit ins Spiel: Produktgüte, Produktkomplexität, Plattformkomplexität.

COCOMO II – Early design level

$$PM_{NS} = A \cdot \mathrm{KLOC}^{E} \cdot EM + \mathrm{PM_m}$$

Korrekturfaktor $\mathrm{PM_m}$ bei viel generiertem Code

höhere Produktivität bei Code-Generierung; nicht weiter diskutiert hier.

Faktoren für Exponent E I

$$PM_{NS} = A \cdot \text{KLOC}^{E} \cdot EM + \text{PM}_{\text{m}}$$

- Erfahrung mit Anwendungsbereich (PREC)
  - → Erfahrung mit vorliegendem Projekttyp
  - 5 keine Erfahrung
  - 0 vollständige Vertrautheit
- Entwicklungsflexibilität (FLEX)
  - → Grad der Flexibilität im Entwicklungsprozess
  - 5 Prozess vom Kunden fest vorgegeben
  - 0 Kunde legt nur Ziele fest
- Risikomanagement (RESL)
  - → Umfang der durchgeführten Risikoanalyse
  - 5 keine Risikoanalyse
  - 0 vollständige und genaue Risikoanalyse

Faktoren für Exponent E II

- Teamzusammenhalt (TEAM)
  - → Vertrautheit und Eingespieltheit des Entwicklungsteams
  - 5 schwierige Interaktionen
  - 0 integriertes und effektives Team ohne Kommunikationsprobleme
- Prozessreife (EPML)
  - → Reife des Entwicklungsprozesses (z.B. CMM);
  - → beabsichtigt: gewichteter Anteil der mit „ja“ beantworteten Fragen im CMM-Fragebogen
  - → pragmatisch: CMM-Level
  - 5 niedrigster CMM-Level
  - 0 höchster CMM-Level

Effort Multiplier RCPX: Product Reliability and Complexity

$$PM_{NS} = A \cdot \text{KLOC}^{E} \cdot EM + \text{PM}_{\text{m}} \text{ mit } EM = \prod Effort\text{-}Multiplier_{\text{i}}$$

RELY	Required reliability
DOCU	Documentation match to life-cycle needs
CPLX	Product complexity
DATA	Data base size

Grad:	very low	low	nominal	high	very high	extra high
Punkte:	1	2	3	4	5	6
RCPX						
RELY	very little	little	some	basic	strong	
DOCU	very little	little	some	basic	strong	
CPLX	very little	little	some	basic	strong	very strong
DATA		small	moderate	large	very large	

$\sum$ Punkte:	5–6	7–8	9–11	12	13–15	16–18	19–21
EM_{RPCX}	0,49	0,60	0,83	1,00	1,33	1,91	2,72

DATA: This cost driver attempts to capture the effect large test data requirements have on product development. The rating is determined by calculating D/P, the ratio of bytes in the testing database to SLOC in the program. The reason the size of the database is important to consider is because of the effort required to generate the test data that will be used to exercise the program. In other words, DATA is capturing the effort needed to assemble and maintain the data required to complete test of the program.

Effort Multiplier PDIF: Platform Difficulty

TIME	Execution time constraints (Auslastung CPU)
STOR	Main storage constraints (Auslastung RAM)
PVOL	Platform volatility (Häufigkeit von Plattformänderungen)

Grad:	low	nominal	high	very high	extra high
Punkte:	2	3	4	5	6
PDIF					
TIME		≤50%	≤65%	≤80%	≤90%
STORE		≤50%	≤65%	≤80%	≤90%
PVOL	very stable	stable	somewhat volatile	volatile	

$\sum$ Punkte:	8	9	10–12	13–15	16–17
EM_{PDIF}	0,87	1,00	1,29	1,81	2,61

Effort Multiplier PERS: Personnel Capability

ACAP	Analyst capability (gemessen als Perzentil)
PCAP	Programmer capability (gemessen als Perzentil)
PCON	Personnel continuity (gemessen durch Personalfluktuation)

Grad:	very low	low	nominal	high	very high
Punkte:	1	2	3	4	5
PERS					
ACAP	15%	35%	55%	75%	90%
PCAP	15%	35%	55%	75%	90%
PCON	48%	24%	12%	6%	3%

$\sum$ Punkte:	3–4	5–6	7–8	9	10–11	12–13	14–15
EM_{PERS}	2,12	1,62	1,26	1,00	0,83	0,63	0,50

A percentile rank is the proportion of scores in a distribution that a specific score is greater than or equal to. For instance, if you received a score of 95 on a math test and this score was greater than or equal to the scores of 88% of the students taking the test, then your percentile rank would be 88. You would be in the 88th percentile.

Effort Multiplier PREX: Personnel Experience

AEXP	Applications experience
PLEX	Platform experience
LTEX	Language/tool experience

Grad:	very low	low	nominal	high	very high
Punkte:	1	2	3	4	5
PREX					
AEXP	$\leq$2 Mo.	6 Mo.	1 J.	3 J.	6 J.
PLEX	$\leq$2 Mo.	6 Mo.	1 J.	3 J.	6 J.
LTEX	$\leq$2 Mo.	6 Mo.	1 J.	3 J.	6 J.

$\sum$ Punkte:	3–4	5–6	7–8	9	10–11	12–13	14–15
EM_{PREX}	1,59	1,33	1,22	1,00	0,87	0,74	0,62

Effort Multiplier FCIL: Facilities

TOOL	Use of software tools
SITE	Multisite development

Grad:	very low	low	nominal	high	very high		
Punkte:	1	2	3	4	5	6	
FCIL							
TOOL	(1)	(2)	(3)	(4)	(5)		→ nächste Folie
SITE	(1)	(2)	(3)	(4)	(5)	(6)	→ nächste Folie

$\sum$ Punkte:	2	3	4–5	6	7–8	9–10	11
EM_{FCIL}	1,43	1,30	1,10	1,00	0,87	0,73	0,62

TOOL und SITE

TOOL:

1. Editor, Compiler, Debugger
2. einfaches CASE-Werkzeug, schlechte Integration
3. Basis-Life-Cycle-Tools moderat integriert
4. weitergehende, reife Life-Cycle-Tools moderat integriert
5. weitergehende, proaktive Life-Cycle-Tools gut integriert mit Prozessen, Methoden und Wiederverwendung

SITE:

1. Telefon prinzipiell vorhanden und Post
2. individuelles Telefon und Fax
3. E-Mail (niedrige Bandbreite)
4. elektronische Kommunikation mit großer Bandbreite
5. elektronische Kommunikation mit großer Bandbreite, gelegentliche Videokonferenzen
6. interaktive Multimedia

Effort Multiplier SCED: Schedule

Es besteht Notwendigkeit, den Zeitplan zu straffen bzw. es wird mehr Zeit als notwendig eingeräumt.

SCED = Verkürzung bzw. Verlängerung des nominalen Zeitplans.

	75%	85%	100%	130%	160%
EM_{SCED}	1,43	1,14	1,00	1,00	1,00

This rating measures the schedule constraint imposed on the project team developing the software. The ratings are defined in terms of the percentage of schedule stretch-out or acceleration with respect to a nominal schedule for a project requiring a given amount of effort. Accelerated schedules tend to produce more effort in the later phases of development because more issues are left to be determined due to lack of time to resolve them earlier. A schedule compress of 74% is rated very low. A stretch-out of a schedule produces more effort in the earlier phases of development where there is more time for thorough planning, specification and validation. A stretch-out of 160% is rated very high.
Stretch-outs (i.e., SCED $>$ 100) do not add or decrease effort. Their savings bacause of smaller team size are generally balanced by the need to carry project administrative functions over a longer period of time.

Rechenbeispiel

Nominaler Aufwand

Personenmonate $PM_{NS} = A \cdot \text{KLOC}^{E} \cdot EM$ mit $EM_{SCED} = 1,0$

mit $A = 2,94$ und $E = B + \sum w_i/100$ mit $B = 0,91$.

Annahme: es herrschen einfache Verhältnisse:

- $\rightarrow \forall i : w_i = 0 \Rightarrow E = 0,91$ (bester Fall)
- $\rightarrow$ nominale Effort-Multiplier = 1,00 (Normalfall) $\Rightarrow EM = 1,00$

Geschätzte Programmlänge = 100 [KLOC]

$\rightarrow PM_{NS} = 2,94 \times 100^{0,91} \times 1,0 = 194,24 \text{ Monate } \approx 16 \text{ Jahre}$

Entwicklungsdauer

Nominale Entwicklungsdauer (Kalenderzeit in Monaten)

$TDEV_{NS} = C \times PM_{NS}^{D+0,2\times(E-B)}$

mit $C = 3,67$ und $D = 0,28$.

Beispiel: $TDEV_{NS} = 3,67 \times 194,24^{0,28+0,2\times(0,91-0,91)} \approx 16$

Anzahl Entwickler

$N = PM_{NS}/TDEV_{NS}$

Beispiel: $N = 194,24/16 = 12$

Verkürzte Entwicklungsdauer

Chef: „Wieso 16 Monate? Geht das nicht schneller?“

PM_{NS} geht von $SCED = 1,0$ aus.

Abweichung von der nominalen Entwicklungdauer

$TDEV = TDEV_{NS} \times SCED$

Wir verkürzen auf 75%:

$TDEV = 16 \times 75/100 = 12$ Monate

Chef: „Super!“

Wir setzen $SCED = 75$ in PM-Formel ein.

$PM = 2,94 \times 100^{0,91} \times 1,0 \times 1,43 = 277,76$

Erhöhung des Aufwands: um 43 %

Chef: „43% mehr Kosten? Seid Ihr wahnsinnig?“

COCOMO II – Post-architecture level

Berücksichtigt:

- Auswirkungen erwarteter Änderungen von Anforderungen
- Ausmaß/Aufwand der möglichen Wiederverwendung
  - Aufwand für Entscheidung, ob Wiederverwendung
  - Aufwand für das Verstehen existierenden Codes
  - Aufwand für Anpassungen
- 17 verfeinerte lineare Einflussfaktoren
- Schätzung basiert auf LOC

COCOMO II – Post-architecture level

Produktgüte/-kompl	→	Verlässlichkeit, Datenbasisgröße, Komplexität, Dokumentation
Plattformkomplexität	→	Laufzeit-, Speicherbeschränkungen, Plattformdynamik
Fähigkeiten Personal	→	Fähigkeiten der Analysten/Entwickler, Kontinuität des Personals
Erfahrung Personal	→	Domänenerfahrung der Analysten/Entwickler, Erfahrung mit Sprache und Werkzeugen
Infrastruktur	→	Tools, verteilte Entwicklung+Kommunikation

Einflussfaktoren (Cost Drivers) für Cocomo-2

	- -	-	o	+	++	+++
Product Attributes						
RELY – Required reliability	0,82	0,92	1,00	1,10	1,26	
DATA – Data base size		0,90	1,00	1,14	1,28	
CPLX – Product Complexity	0,73	0,87	1,00	1,17	1,34	1,74
RUSE – Required Reusability		0,95	1,00	1,07	1,15	1,24
DOCU – Doc, match to life-cycle needs	0,81	0,91	1,00	1,11	1,23	
Computer Attributes						
TIME – Execution time constr.			1,00	1,11	1,29	1,63
STOR – Main storage constr.			1,00	1,05	1,17	1,46
PVOL – Platform volatility		0,87	1,00	1,15	1,30	

Einflussfaktoren (Cost Drivers) für Cocomo-2

	- -	-	o	+	++	+++
Personell attributes						
ACAP – Analyst capability	1,42	1,19	1,00	0,85	0,71	
PCAP – Programmer capability	1,34	1,15	1,00	0,88	0,76	
AEXP – Applications experience	1,22	1,10	1,00	0,88	0,81	
PLEX – Platform experience	1,19	1,09	1,00	0,91	0,85	
LTEX – Language/tool exp.	1,20	1,09	1,00	0,91	0,84	
Project attributes						
TOOL – Use of software tools	1,17	1,09	1,00	0,90	0,78	
SITE – Multisite development	1,22	1,09	1,00	0,93	0,86	0,80
SCED – Required dev. schedule	1,43	1,14	1,00	1,00	1,00	

Zusammenfassung

- alle Schätzungen basieren auf Erfahrung
- kontinuierlich schätzen
- verschiedene Techniken anwenden

Weiterführende Literatur

- Poensgen und Bock (2005) fassen auf 160 Seiten das Wichtigste zur Function-Point-Analyse zusammen. Hilfreich für das Verständnis ist ein umfangreiches Beispiel, bei dem die Unadjusted Function Points für ein Microsoft-Adressbuch ermittelt werden.
- Das Buch von Boehm u. a. (2000) ist die Referenz für COCOMO II; das Buch ist sehr umfassend und detailliert; es beschreibt Praxisbeispiele und die Kalibrierung des Modells mit neuen Daten.

Wiederholungs- und Vertiefungsfragen I

- Welche Möglichkeiten zur Schätzung von Aufwand und Kosten für die Software-Entwicklung gibt es?
- Wann wird geschätzt?
- Erläutern Sie die Function-Point-Methode (am konkreten Beispiel).
- Was sind Adjusted Function-Points im Unterschied zu Unadjusted-Function-Points?
- Wie errechnet sich der Aufwand aus den Function-Points?
- Was ist die Idee von Object-Points im Gegensatz zu Function-Points?
- Erläutern Sie die CoCoMo-Methode (neue Version) für die Level *Early Prototyping* und *Early Design Level*.
- Geben Sie die grundsätzliche Formel für den Aufwand wieder. Was bedeuten die verschiedenen Parameter?

Wiederholungs- und Vertiefungsfragen II

- Um welche Art von Parametern handelt es sich bei den Faktoren, die im Exponenten auftreten?
- Wozu die Unterscheidung in die verschiedene Stufen (Level)?
- Wie errechnet sich die Entwicklungsdauer aus dem Aufwand?
- Wie wird eine Verkürzung der nominalen Entwicklungsdauer behandelt?
- Vergleichen Sie die Function-Point-Methode mit CoCoMo.

N.B.:

1. Die Übungsaufgaben sind weitere Beispiele relevanter Fragen.
2. Bei der Darstellung der Einflussfaktoren für die Adjusted Function Points genügt es, ein paar Beispiele nennen zu können und zu wissen, worauf sie sich grundsätzlich beziehen (System und nicht Entwicklungsteam/-prozess); keinesfalls wird Gedächtnisleistung abgeprüft; das gilt auch für die Einflussfaktoren von CoCoMo.

Wiederholungs- und Vertiefungsfragen III

3. Bei der Darstellung von Function-Points und CoCoMo interessieren nicht die absoluten Zahlen, aber sehr wohl der grundsätzliche Aufbau der Formeln.

Empirische Softwaretechnik

Empirische Softwaretechnik
- Motivation
- Wissenserwerb in Wissenschaft und Engineering
- Untersuchungsmethoden
- Bestandteile eines Experiments

Lernziele

- die Notwendigkeit zur empirischen Forschung in der Softwaretechnik erkennen
- prinzipielles Vorgehen verstehen
- (irgendwann einmal) empirisch forschen können

Experimente in der Softwaretechnik

> *Experimentation in software engineering is necessary but difficult. Common wisdom, intuition, speculation, and proofs of concept are not reliable sources of credible knowledge.*
>
> – V.R. Basili, 1999

Motivation

- Wir wollen genau wissen, ob und unter welchen Randbedingungen eine Methode funktioniert.
- Forschung
  - beweist durch logische Schlüsse
  - oder aber beobachtet, experimentiert und misst.
- Messung ist sorgfältige Beobachtung mit größtmöglicher Präzision, Zuverlässigkeit und Objektivität
- Messungen identifizieren neue Phänomene, testen Hypothesen oder leiten uns bei der Anwendung von Modellen und Methoden

Empirische Untersuchungen

Methoden, die von Menschen angewandt werden, können nur empirisch untersucht werden.

Beispiele

Experiment von Knight und Leveson (1986):

Hypothese: N-Version-Programming führt zu zuverlässigerer Software.

Zugrundeliegende Annahme:
Unabhängigkeit der Einzelwahrscheinlichkeiten, d.h.
Gesamtfehlerwahrscheinlichkeit P eines N-Versionen-Programms ist das Produkt der Fehlerwahrscheinlichkeiten aller N Versionen

N = 27 Versionen, 1 Million Testfälle

- alle N zuverlässiger als 99 Prozent
- 23 sogar zuverlässiger als 99,9 Prozent
- 6 funktionierten ganz fehlerfrei

Beobachtete Gesamtfehlerwahrscheinlichkeit liegt über P.

Knight and Leveson's experiment analyzed the failure probabilities of multiversion programs. Conventional theory predicted that the failure probability of a multiversion program was the product of the failure probabilities of the individual versions. However, John Knight and Nancy Leveson observed that real multiversion programs had significantly higher failure probabilities. In essence, the experiment falsified the basic assumption of the conventional theory, namely that faults in program versions are statistically independent.

– Tichy (1998)

Details: `http://page.mi.fu-berlin.de/prechelt/swt2/node28.html`

Beispiele

Experiment von Dzidek u. a. (2008):

Hypothese: Dokumentation in UML hilft bei der Wartung.

Versuchsaufbau:

- Kontrollgruppe hat User-Manual, Grobbeschreibung der Pakete und ihre Relation zu anderen Paketen, verwendete Bibliotheken, Datenbankschema, Dokumentation fürs Deployment sowie Javadoc-Kommentare.
- Experimentalgruppe hat zusätzlich UML-Dokumentation.

Ergebnisse:

- signifikante 54 % Verbesserung funktionaler Korrektheit der Änderungen ($p = 0,03$)
- insignifikante 7 % Verbesserung der Entwurfsqualität ($p = 0,22$)
- insignifikante 14 % Erhöhung der Entwicklungszeit verursacht durch die Aktualisierung der UML-Dokumentation ($p = 0,35$)

This is the first controlled experiment that investigates the costs of maintaining and the benefits of using UML documentation during the maintenance and evolution of a real nontrivial system, using professional developers as subjects, working with a state-of-the-art UML tool during an extended period of time. The subjects in the control group had no UML documentation. In this experiment, the subjects in the UML group had, on average, a practically and statistically significant 54 percent increase in the functional correctness of changes (p=0.03) and an insignificant 7 percent overall improvement in design quality (p=0.22), though a much larger improvement was observed on the first change task (56 percent), at the expense of an insignificant 14 percent increase in development time caused by the overhead of updating the UML documentation (p=0.35).
Design quality Q is measured in terms of following proper OO design principles. This was calculated by first breaking each task into subtasks and rating each as either acceptable or unacceptable according to the predefined criteria. Specifically, a solution to a subtask with one of the following problems would be deemed as unacceptable

- duplication (copying and pasting) of existing code instead of direct use of that logic (e.g., copying and pasting a sort method),
- addition of new design elements that current design elements could have handled (e.g., creating a new partial user class even though an existing user class is present),
- incorrect placement of logic in a class,
- distribution of logic throughout the application when adding it to just one place would have had the same effect, and
- use of the try/catch mechanism as a normal part of the application's logic.

Q counts the number of acceptable subtask solutions.

– Dzidek u. a. (2008)

Beispiel von Müller (2006) I

Objective:
Comparison of program defects caused by programmer pairs and solo developers.

Design:
Analysis of programs developed during two counter balanced experiments.

Setting:
Programming lab at University.

Experimental Units:
42 programs developed by computer science students participating in an extreme programming lab course.

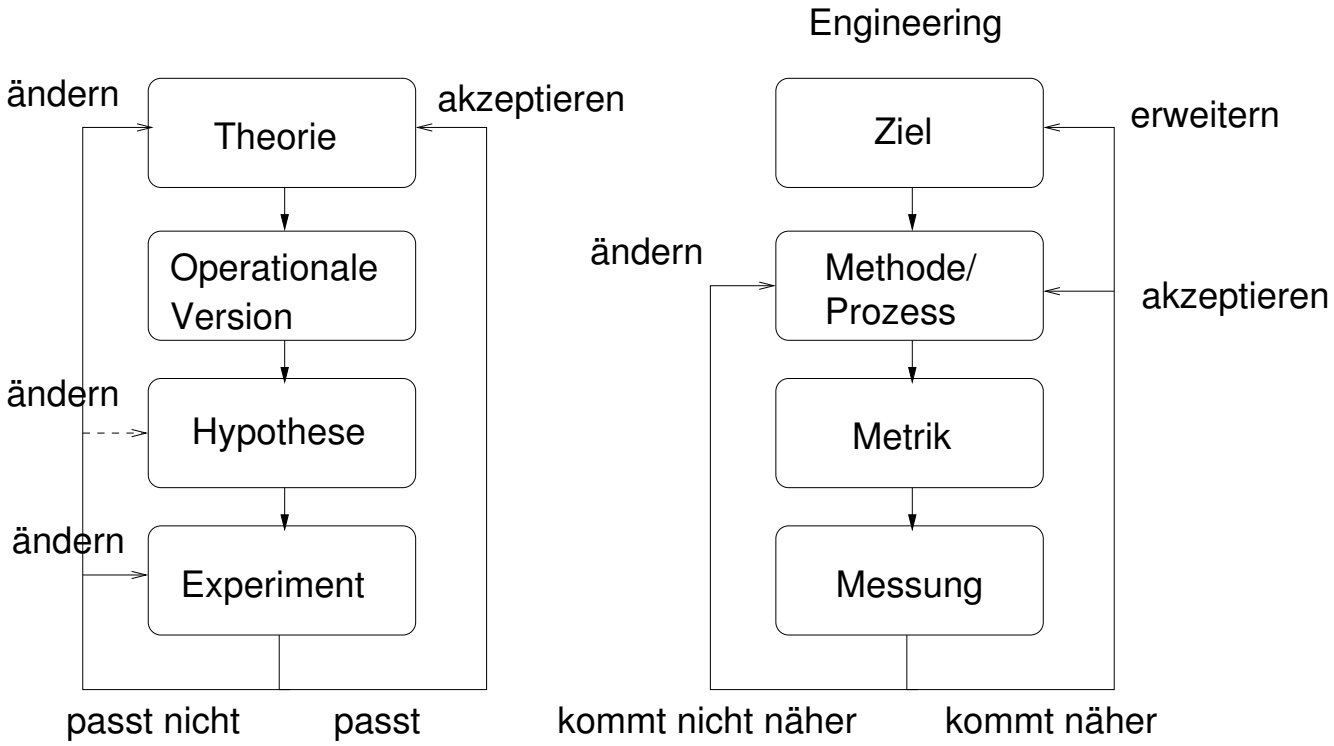
Main Outcome Measures:
Programmer pairs make as many algorithmic mistakes but fewer expression mistakes than solo programmers.

Beispiel von Müller (2006) II

Results: The second result is significant on the 5 percent level.

Conclusions: For simple problems, pair programming seems to lead to fewer mistakes than solo programming.

Wissenserwerb in Wissenschaft und Engineering



Beschaffenheit empirischer Forschung

- in der Softwaretechnik erst seit ungefähr 1980 als wesentliche Disziplin anerkannt
- heute: Konferenzen und Zeitschriften
- Verschiebung weg von rein mathematischen Methoden
- Mehrzahl der Probleme der Softwaretechnik sind nicht mathematischer Art, hängen vielmehr von Menschen ab

Untersuchungsmethoden

- Umfragen und Erhebungen
- Geschichte
- Archivarische Analyse
- Fallstudien
- kontrollierte Experimente

Untersuchungsmethoden

Umfragen und Erhebungen:

- Datenerfassung mit Fragebögen oder ethnographische Studien
- können auch a-posteriori durchgeführt werden
- dienen oft der Bildung von Hypothesen für nachfolgende Fallstudien oder kontrollierte Experimente
- fundierte Methoden zur Datenerhebung und -validierung notwendig
- entstammen den Sozialwissenschaften und kognitiven Wissenschaften

Untersuchungsmethoden

Definition einer Fallstudie nach Yin (2003):

A case study is an empirical inquiry that

- *investigates a contemporary phenomenon within its real-life context, especially when*
- *the boundaries between phenomenon and context are not clearly evident*

The case study inquiry

- *copes with the technically distinctive situation in which there will be many more variables of interest than data points, and as one result*
- *relies on multiple sources of evidence, with data needing to converge in a triangulating fashing, and as another result*
- *benefits from the prior development of theoretical propositions to guide data collection and analysis.*

Untersuchungsmethoden

Fallstudien (Quasi-Experimente):

- In-Vivo-Experiment oder Feldstudien mit Hypothese
- eingebettet in echte Projekte und damit weniger kontrolliert
- Herausforderung: zu messen, ohne den Verlauf zu verfälschen

Untersuchungsmethoden

kontrollierte Experimente (In-Vitro-Experiment):

- finden in kontrollierter Testumgebung statt
- Hypothese wird falsifiziert oder bestätigt (mit einer gewissen Wahrscheinlichkeit)
- unabhängige Variablen: Eingabeparameter, die im Experiment kontrolliert (variiert) werden
- abhängige Variablen: Ausgabeparameter, die gemessen werden

Übersicht über Untersuchungsmethoden

	Art der Frage	mögliche Kontrolle	Fokus auf Gegenwart
Experimente	wie, warum?	ja	ja
Umfragen und Erhebungen	wer, was, wo, wie viele, wie sehr?	nein	ja
Archivarische Analyse	wer, was, wo, wie viele, wie sehr?	nein	ja/nein
Geschichte	wie, warum?	nein	nein
Fallstudien	wie, warum?	nein	ja

Quelle: COSMOS Corporation, erläutert von Yin (2003)

Geschichtliche Analyse betrachtet Ereignisse, die in der Vergangenheit liegen. Die angewandten Techniken können mit denen einer Fallstudie überlappen. Es ist aber im Unterschied zu Fallstudien nicht möglich, die Geschehnisse direkt zu beobachten (sie liegen in der Vergangenheit) und die Beteiligten zu befragen (es gibt sie nicht mehr). Die historische Analyse stützt sich auf primäre und sekundäre Literatur sowie kulturelle und physische Artefakte.

Bestandteile eines Experiments

1. Festlegung der Ziele

Aspekte:

1. Objekt der Studie (z.B. Entwurf, Codierung, Test)
2. Zweck der Studie (z.B. Vergleich, Analyse, Voraussage)
3. Fokus der Studie (z.B. Effektivität, Effizienz)
4. Standpunkt (z.B. Praktiker, Forscher)
5. Kontext (z.B. Erfahrung der Teilnehmer, verwendete Elemente, Umgebung)

Kontext $\rightarrow$ unabhängige Variablen
Fokus $\rightarrow$ abhängige Variablen

Bestandteile eines Experiments

2. Formulierung einer testbaren Hypothese

„Methode M ist geeignet für Aufgabe A“
versus
„Methode M benötigt weniger Zeit als Methode N, um Aufgabe A zu erledigen“

Null-Hypothese (Negation der Hypothese):
„Methode M benötigt die gleiche oder mehr Zeit als N, um Aufgabe A zu erledigen“

Wenn Null-Hypothese widerlegt ist, wird Hypothese bestätigt (mit einem gewissen Grad an Konfidenz)

Bestandteile eines Experiments

3. Aufbau des Experiments

- Korrelation versus Kausalität
- Identifikation aller unabhängiger und abhängiger Variablen
- Validität
  - interne: es werden tatsächlich nur die Beziehungen zwischen unabhängigen und abhängigen Variablen gemessen (z.B. keine Lerneffekte, zeitliche Aspekte, Auswahl der Teilnehmer)
  - externe: Übertragbarkeit (z.B. Studenten versus Praktiker)
- Randomisierung

$\rightarrow$ viele Bücher zu Standard-Designs abhängig von den Rahmenbedingungen

Bestandteile eines Experiments

4. Analyse und Validierung der Resultate

- Auswertung durch statistische Methoden (Korrelationen und Regressionsanalyse)
- Problem des geringen Datenumfangs
  - parametrische statistische Tests setzen bestimmte Verteilung voraus
  - meist nur nicht-parametrische statistische Tests adäquat, weil keine Verteilung voraus gesetzt wird (insbesondere für Nominal- bis Ordinalskalen)
- quantitative Analyse oft ergänzt durch qualitative
- Validierung
  - Befragungen der Teilnehmer, um sicher zu stellen, dass sie alle Fragen verstanden haben
  - Untersuchung statistischer Ausreißer
- Benchmarking schwierig, weil Firmen ihre Daten ungern veröffentlichen

Bestandteile eines Experiments

5. Replikation

- Wiederholung, um „Glückstreffer“ auszuschließen
- Experiment muss detailliert beschrieben sein
- bedauerlicherweise schwer zu veröffentlichen, weil keine neuen Ergebnisse präsentiert werden

Ethische Fragen

- keine Teilnahme ohne explizite Einwilligung
- kein Missbrauch
- Anonymität muss gewährt werden (doppelt blind)
- kein materieller oder sonstiger Gewinn (Bezahlung, Gehaltserhöhung, gute Note etc.) oder Verlust

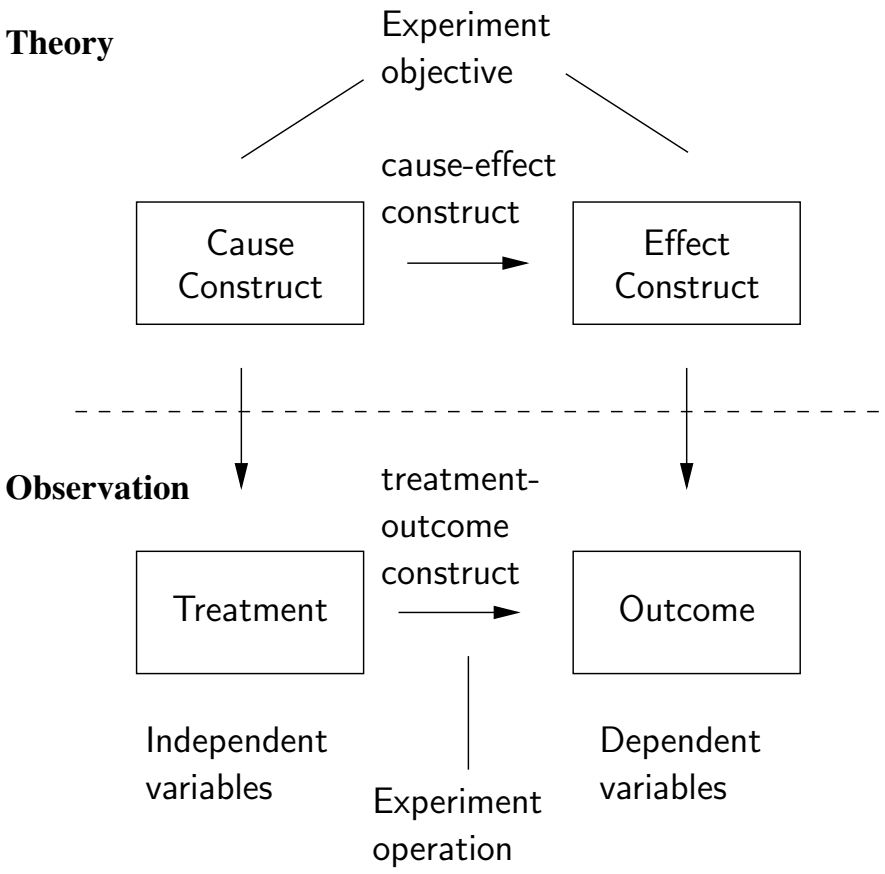
Grenzen der experimentellen Forschung

- hoher Aufwand (allerdings: Lernen ohne Experimente ist auch teuer)
- Abhängigkeit von menschlichen Versuchsteilnehmern
  - Studenten haben möglicherweise nicht die Erfahrung
  - Praktiker haben wenig Zeit
- Transferierbarkeit der Resultate
  - Software-Projekte haben eine Unzahl möglicher Variablen
  - nicht alle sind kontrollierbar
  - und wenn sie kontrolliert sind, treffen sie möglicherweise nicht auf andere Umgebungen zu

Kontrollierte Experimente

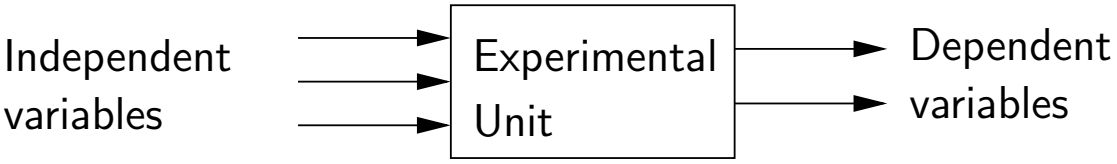
Kontrollierte Experimente
- Begriffe
- Auswahl der Teilnehmer
- Aufbau von Experimenten
- Gültigkeit (Threats to Validity)
- Wiederholungsfragen

Theorie und Beobachtung (Wohlin u. a. 2000)

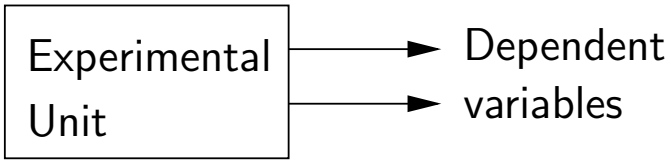


Experiment

Experiment



Observational Study



Definition

Experiment: An investigation in which the investigator applies some treatments to experimental units and then proceeds to observe the effect of the treatments on the experimental units by measuring one or more dependent (response) variables.

Definition

Observational Study: an investigation in which the investigator observes units and measures one or more response variables without imposing treatments on the individuals.

Zieldefinition eines Experiments

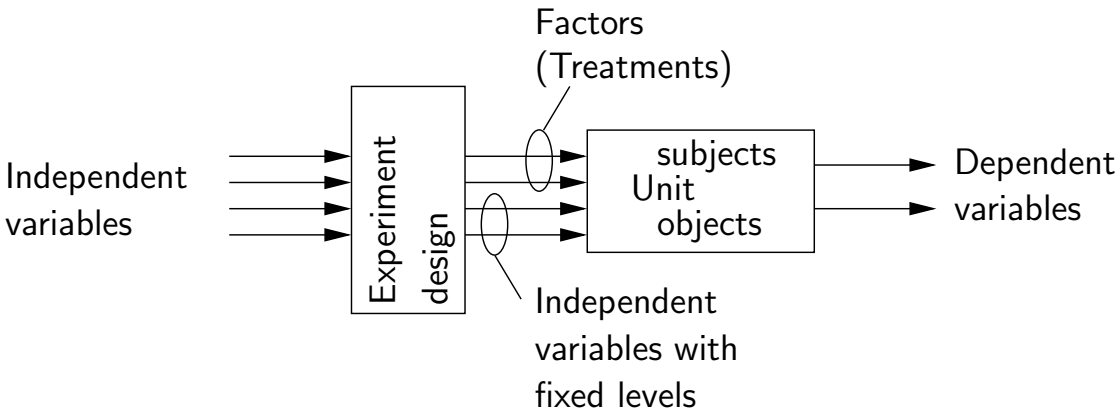
Vorlage:

Analyse *Object of Study*
for the purpose of *Purpose*
with respect to *Quality focus*
from the point of view of the *Perspective*
in the context of *Context*.

Beispiel:

Analyse *Quality assurance process*
for the purpose of *Characterize code inspection onto quality*
with respect to *Effectiveness and Cost*
from the point of view of the *Researcher*
in the context of *Professional developers in a software organization*.

Experiment



Definition

Independent Variable: variable that can be controlled and changed in the experiment.

z.B. Code-Inspektion, Erfahrung der Gutachter, inspizierte Software.

Definition

Factor: an explanatory variable studied in an investigation that can be set at any one of two or more values.

z.B. Code-Inspektion

Definition

Levels: the different values of a factor.

z.B. Code-Inspektion: angewandt oder nicht

Definition

Treatment: the circumstances created for an experiment, i.e., one particular set of values of factors.

z.B. Code-Inspektion wird angewandt

Definition

Object: the entity that receives the treatment.

z.B. die Software

Definition

Subject (Participant): the person that applies the treatment.

z.B. Entwickler

Definition

Test (Trial): a combination of treatment, subject, and object. An experiment consists of a set of tests.

z.B. Entwickler wendet Code-Inspektion für Software an.

Definition

Dependent Variable (Response Variable): a characteristic of an experimental unit measured after treatment and analyzed to address the objectives of the experiment.

z.B. Anzahl gefundener Fehler und Dauer

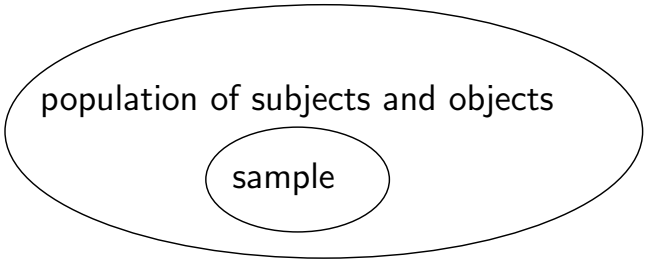
Definition

Experimental Error: accounts for the fact that experimental units treated independently and identically will not have identical dependent variable measurements.

z.B. Tagesform der Entwickler führt zu unterschiedlichen Messungen.

Auswahl der Teilnehmer

Sampling: Auswahl/Stichprobe von Objects und Subjects



Aspekte:

- Auswahlgröße beeinflusst experimentellen Fehler und Aussagekraft des statistischen Tests
- je höher die Varianz in der Population, desto größer muss die Auswahl sein
- Art der späteren Analyse beeinflusst Auswahlgröße

→ Art der Analyse frühzeitig festlegen

Auswahl der Teilnehmer

Sampling-Strategien:

- Quasi-Experiment: Subjects nicht zufällig ausgewählt
- Convenience Sampling: Auswahl nach Einfachheit
- Simple Random Sampling: zufällige Auswahl
- Systematic Sampling: Population wird angeordnet; erstes Subject zufällig gewählt, dann jedes n'te Subject

→ Annahme: Population der Subjects ist homogen

Auswahl der Teilnehmer

Partitionierende Sampling-Strategien:
Partitionierung der Population in homogene Gruppen

- Quota Sampling: feste Quota für die Auswahl aus Partitionen ist vorgegeben, dann Convenience Sampling für einzelne Gruppen
- Stratified Random Sampling
  - Proportionate Stratified Random Sampling: zufällige Auswahl aus jeder Gruppe ist proportional zur Gesamtpopulation; Bsp.: 60 % Männer, 40 % Frauen $\rightarrow$ 3 Männer, 2 Frauen
  - Optimum (Disproportionate) Stratified Random Sampling: zufällige Auswahl aus Gruppe ist proportional zur Standardabweichung der Verteilung der Variable (mehr Auswahlen für die Gruppen mit höherer Verschiedenheit)

Generelle Prinzipien

- Randomization: Beobachtungen betreffen unabhängige Zufallsvariablen (Zuordnung Treatments–Subjects–Objects und Reihenfolge der Treatments)
- Blocking:
  1. Ähnliche Objekte werden gruppiert
  2. Treatments werden durch Vergleich der abhängigen Variablen desselben Blocks evaluiert
- Balancing: jedes Treatment hat gleiche Anzahl von Subjects
- Replication: mindestens ein Treatment wird unabhängig auf zwei oder mehr Objekte angewandt

Experimental Design

Arten von Studien:

		# objects	
		1	**>1**
# subjects per object	**1**	single-object	multi-object variation
	>1	multi-test within object	blocked subject-object

Standard-Designs

Definition

Full Factorial Treatment Design: the treatments consist of all possible combinations of the levels of the factors of interest.

- ein Faktor mit zwei Treatments
- ein Faktor mit mehr als zwei Treatments
- zwei Faktoren mit zwei Treatments
- mehr als zwei Faktoren mit mehr als zwei Treatments

Ein Faktor mit zwei Treatments

Definition

Completely randomized design: alle möglichen Zuordnungen von Treatments und Subjects/Objects sind gleich wahrscheinlich.

Subjects	Treatment 1	Treatment 2
1	X	
2		X
3		X
4	X	
5	X	
6		X

μ_i = Mittelwert der abhängigen Variablen für Treatment i
Nullhypothese: $\mu_1 = \mu_2$
Statistische Tests: t-test, Mann-Whitney

Ein Faktor mit zwei Treatments

Definition

Paired comparison design: jedes Subject wendet alle Treatments an. Reihenfolge wird randomisiert.

Subjects	Treatment 1	Treatment 2
1	1	2
2	2	1
3	2	1
4	1	2
5	1	2
6	2	1

y_{ij} = j'te Messung der abhängigen Variablen für Treatment i
$d_j = y_{1j} - y_{2j}$ und μ_d = Mittelwert von d
Nullhypothese: $\mu_d = 0$
Statistische Tests: Paired t-test, Sign test, Wilcoxon

Ein Faktor mit mehr als zwei Treatments

Completely randomized design

Subjects	Treatment 1	Treatment 2	Treatment 3
1	X		
2		X	
3			X
4	X		
5			X
6		X	

Nullhypothese: $\mu_1 = \mu_2 = \cdots = \mu_n$
Statistische Tests: ANOVA, Kruskal-Wallis

Ein Faktor mit mehr als zwei Treatments

Paired comparison design

Subjects	Treatment 1	Treatment 2	Treatment 3
1	1	2	3
2	1	3	2
3	2	1	3
4	2	3	1
5	3	1	2
6	3	2	1

Nullhypothese: $\mu_1 = \mu_2 = \cdots = \mu_n$
Statistische Tests: ANOVA, Kruskal-Wallis

Zwei Faktoren

Definition

2 × 2 Factorial Design: zufällige Zuordnung der Subjects für jede Kombination der Treatments

		Faktor A	
		Treatment A1	Treatment A2
Faktor B	Treatment B1	4, 6	1, 7
	Treatment B2	2, 3	5, 8

α_i = Effekt von Treatment i auf Faktor A
β_i = Effekt von Treatment i auf Faktor B
$(\alpha\beta)_{ij}$ = Effekt der Interaktion von α_i und β_i
Nullhypothesen: $\alpha_1 = \alpha_2$ oder $\beta_1 = \beta_2$ oder $(\alpha\beta)_{ij} = 0$ für alle i, j
Statistische Tests: ANOVA

Tabelleninhalt beschreibt die durchnummerierten Subjects $1 - 8$.

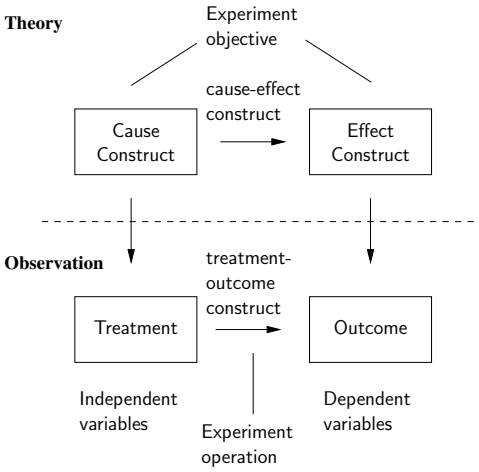
Mehr als zwei Faktoren mit zwei Treatments

Definition

2^k **Factorial Design für k Faktoren**: zufällige Zuordnung der Subjects für jede der 2^k Kombinationen der Treatments

Faktor A	Faktor B	Faktor C	Subjects
A1	B1	C1	2, 3
A2	B1	C1	1, 13
A1	B2	C1	5, 6
A2	B2	C1	10, 16
A1	B1	C2	7, 15
A2	B1	C2	8, 11
A1	B2	C2	4, 9
A2	B2	C2	12, 14

Gültigkeit (Threats to Validity)



Definition
Conclusion Validity: es gibt einen signifikanten statistischen Zusammenhang von Treatment und Resultat

Definition
Internal Validity: der beobachtete Zusammenhang zwischen Treatment und Resultat ist kausal

Definition
Construct Validity:

1. Treatment reflektiert Construct of Cause
2. Outcome reflektiert den Effekt

Definition
External Validity: das Ergebnis ist auch außerhalb der Studie anwendbar (für andere Personen, Orte, Zeiten etc.)

Internal Validity

- **Störvariable**: zusätzliche unkontrollierte Variable ändert sich systematisch mit den unabhängigen Variablen

  z.B. Pair-Programmer-Teams bestehen nur aus Entwicklern mit großer Erfahrung
- **Historie/Zeit**: äußere Ereignisse beeinflussen Messung

  z.B. Pair-Programmer arbeiten morgens,
  Einzel-Programmierer nachmittags
- **Sättigung**: interne (physische oder psychische) Änderung der Subjects

  z.B. Ermüdung

Internal Validity

- **Wiederholtes Testen**: frühe Treatments beeinflussen nachfolgende Treatments

  z.B. Lerneffekte
- **Instrumentierung**: Veränderung der Art der Messung

  z.B. Fehler bei der Messung mit späterer Korrektur
- **Regression zur Mitte**: bei zwei in irgendeiner Weise verbundenen Messungen gehen extreme Abweichungen bei einer der beiden Messungen im Durchschnitt mit weniger extremen Abweichungen bei der anderen Messung einher

  z.B. Körpergröße von Vätern und Söhnen

Die Regression zur Mitte ist ein Begriff der Statistik; er beschreibt das Phänomen, dass bei zwei in irgendeiner Weise verbundenen Messungen, z. B. Körpergröße eines Vaters und seines Sohnes, extreme Abweichungen bei einer der beiden Messungen im Durchschnitt mit weniger extremen Abweichungen bei der anderen Messung einhergehen – im Beispiel also haben sehr große Väter im Durchschnitt Söhne die weniger groß sind (die aber im Durchschnitt trotzdem noch größer sind als der Durchschnitt der Bevölkerung), oder umgekehrt: sehr große Söhne haben im Durchschnitt Väter die weniger groß sind.

– Quelle: Wikipedia

Internal Validity

- **Selektion**: keine zufällige Zuordnung oder Zuordnung ist zufällig ungünstig

  z.B. die ersten 20 Freiwilligen werden der Pair-Programming-Gruppe zugeordnet
- **Selektionsinteraktion**: Selektions-Threat wird mit anderem internal Threat kombiniert

  z.B. eine Gruppe wird von Programmierern aus Abteilung X dominiert (Selektions-Threat) und Abteilung X hat am Vorabend eine wilde Party (Historie-Threat)

Internal Validity

- **Experimentelle Mortalität**: Unterschiede in der Ausscheiderate über die unterschiedlichen experimentellen Bedingungen

  z.B. in der Pair-Programming-Gruppe fallen mehrere Subjects aus, was die Gruppenzusammensetzung beeinflussen kann
- **Experimentatoreinfluss**: Erwartungen der Durchführenden oder Subjects können Resultat beeinflussen

  z.B. Gruppen werden unbewusst verschieden behandelt oder Subjects versuchen, das vermeintlich gewünschte Ergebnis zu erreichen

External Validity

- **Repräsentanz**: Subjects sind nicht repräsentativ
- **Eignung-Treatment-Interaktion**: Sample hat Eigenschaften, die mit der unabhängigen Variablen interagieren

  z.B. Subjects sind hochmotivierte Freiwillige.
- **Situation**: situationsbedingte (zeitliche/örtliche) Eigenheiten beeinflussen Ergebnis

  z.B. große Displays im Pair-Programming-Experiment
  z.B. Tests finden immer nur morgens statt
- **Reaktivität**: Effekt ist auf die Beobachtung der Situation zurückzuführen

  z.B. Programmierer sind während des Experiments konzentrierter

Wiederholungs- und Vertiefungsfragen I

- Welche Arten von empirischen Untersuchungsmethoden gibt es generell? Was sind ihre Vor- und Nachteile?
- Gegeben sei folgendes Szenario [...] Welche Art von empirischer Untersuchungsmethode würden Sie einschlagen und warum?
- Was sind die wesentlichen Bestandteile eines kontrollierten Experiments? Erläutern Sie diese an einer konkreten Fragestellung.
- Wo liegen die Grenzen empirischer Forschung?
- Was ist der Zusammenhang eines kontrollierten Experiments und einer Theorie?
- Was unterscheidet ein Experiment von einer bloßen Beobachtung?

Wiederholungs- und Vertiefungsfragen II

- Erläutern Sie die Begriffe *abhängige und unabhängige Variablen*, *Faktoren*, *Levels*, *Treatment*, *Objekt* und *Subjekt*, *Test* und *experimenteller Fehler*.
- Erläutern Sie verschiedene Sampling-Methoden und diskutieren Sie ihre Vor- und Nachteile.
- Erläutern Sie die generellen Prinzipien *Randomization*, *Blocking*, *Balancing* und *Replication* eines kontrollierten Experiments.
- Beschreiben Sie verschiedene Versuchsaufbauten in Abhängigkeit der Anzahl der Subjekte und Objekte eines Experiments.
- Beschreiben Sie die Arten von Validitäten einer empirischen Beobachtung. Geben Sie zu jeder Art potentielle Probleme an.

Statistik bei kontrollierten Experimenten

Statistik bei kontrollierten Experimenten
Hypothesen und Stichproben
Verteilungen
Experimente mit einem Sample
Experimente mit zwei Samples
Verteilungsfreier U-Test
Wiederholungsfragen

Hypothese und statistischer Test

Definition

Statistische Hypothese: Aussage über eine statistische Population, die man auf Basis beobachteter Daten zu bestätigen oder zu falsifizieren versucht.

Hypothese:
„Die durchschnittliche Länge von Methoden in Java ist größer als 50 [loc]“

Vorgehen

1. Nimm an, dass die zu testende Hypothese wahr ist.
2. Untersuche die Konsequenzen dieser Annahme in Bezug auf die Sampling-Verteilung, die von der Wahrheit der Hypothese abhängt.

→ Falls die beobachteten Daten eine große Eintrittswahrscheinlichkeit haben, ist die Hypothese bestätigt.

→ Falls die beobachteten Daten eine sehr kleine Eintrittswahrscheinlichkeit haben, gilt die Hypothese als widerlegt.

→ **Signifikanzniveau** α legt die Wahrscheinlichkeit fest, ab der die Hypothese als widerlegt betrachtet wird (konkreter Schwellwert: **kritischer Wert**).

Konvention: $\alpha = 0,05$ oder $\alpha = 0,01$

Nullhypothese und alternative Hypothese

Definition

Nullhypothese H_0: die zu testende Hypothese.
Alternative Hypothese H_1: die Gegenthese zu H_0.

Meist: H_1 ist das, woran der Experimenator wirklich glaubt.
$\rightarrow$ Experiment soll H_0 widerlegen.

Gerichtete und ungerichtete Hypothese

Definition

Ungerichtete Alternativhypothese: Nullhypothese postuliert keinerlei Effekt.

Gerichtete Alternativhypothese: Nullhypothese postuliert keinen oder gegengerichteten Effekt.

Beispiel ungerichtete Alternativhypothese:

- H_1 = Pair-Programming und Single-Programming unterscheiden sich in Qualität.
- H_0 = Pair-Programming und Single-Programming liefern gleiche Qualität.

Beispiel gerichtete Alternativhypothese:

- H_1 = Pair-Programming liefert bessere Qualität als Single-Programming.
- H_0 = Pair-Programming liefert gleiche oder schlechtere Qualität als Single-Programming.

Die Nullhypothese drückt inhaltlich immer aus, dass Unterschiede, Zusammenhänge, Veränderungen oder besondere Effekte in der interessierenden Population überhaupt nicht und/oder nicht in der erwarteten Richtung auftreten. Im Falle einer *ungerichteten* Forschungs- bzw. Alternativhypothese postuliert die Nullhypothese keinerlei Effekt. Im Falle einer *gerichteten* Alternativhypothese geht die Nullhypothese von keinem oder einem gegengerichteten Effekt aus.

– Bortz und Döring (2006)

Hypothesen und Stichproben

Sample = Population $\Rightarrow$ absolute Wahrheit

Sample $\subset$ Population $\Rightarrow$?

Problem:

- Jede Hypothesenüberprüfung liefert statistischen Kennwert (z.B. Durchschnitt) für ein bestimmtes Sample.
- Wiederholung mit anderen Subjects/Objects liefert wahrscheinlich nicht exakt denselben Kennwert.

$\rightarrow$ Kennwert ist Zufallsvariable[3]

- Feststellung, ob Kennwert extrem oder typisch ist, ist ohne Kenntnis der Verteilung der Zufallsvariablen unmöglich.

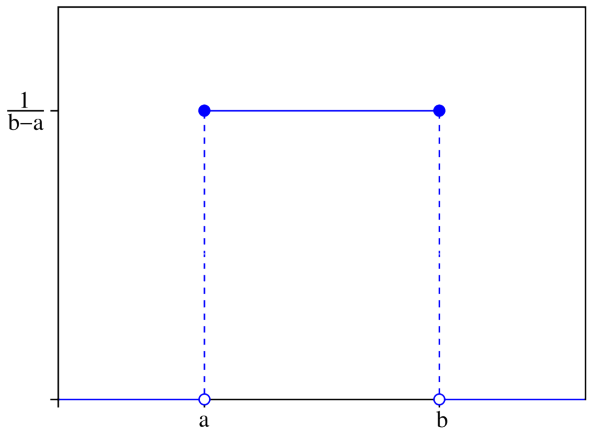
[3] Funktion, die den Ergebnissen eines Zufallsexperiments Werte (so genannte Realisationen) zuordnet.

Verteilungen

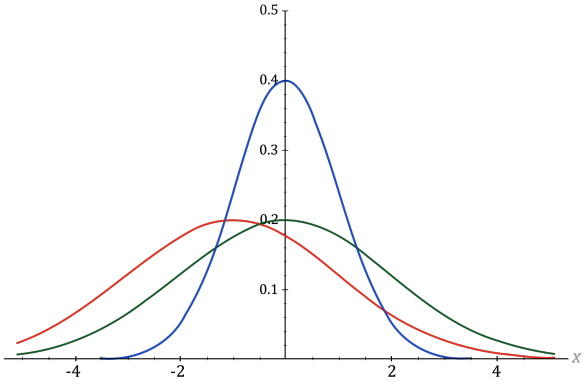
Definition

Verteilung einer Variablen: beschreibt, welche Werte die Variable annehmen kann und wie oft sie das tut.

Gleichverteilung



Normalverteilung



Häufige Kennwerte einer Verteilungen

Gegeben: n Datenpunkte $x_1, x_2, \ldots x_n$ einer Variablen X.

Durchschnitt oder arithmetisches Mittel $\bar{x} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i$

Varianz $s_x^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (x_i - \bar{x})^2$

Standardabweichung $s_x = \sqrt{s_x^2}$

Varianz und Freiheitsgrad

Varianz $s_x^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (x_i - \bar{x})^2$

Warum Durchschnitt mit $\frac{1}{n-1}$?

$\sum_{i=1}^{n} (x_i - \bar{x}) = 0$

- $\rightarrow$ $(x_n - \bar{x})$ kann berechnet werden, wenn $x_1, x_2, \ldots, x_{n-1}$ bekannt sind
- $\rightarrow$ nur $n-1$ Summanden in $\sum_{i=1}^{n} (x_i - \bar{x})^2$ können frei variieren
- $\rightarrow$ $n-1$ ist der **Freiheitsgrad**: Anzahl frei variierbarer Variablen

Der Freiheitsgrad besagt, wie viele der Variablen x_i geändert werden können, so dass die Gleichung $\sum_{i=1}^{n}(x_i - \bar{x}) = 0$ immer noch gilt.
Ein Beispiel: Wir haben die Werte 1,2,3 (also $n = 3$) mit $\bar{x} = 2$. Jetzt ändern wir einen Wert z.B. $1 \rightarrow 0$. Damit aber die Gleichung wieder gilt, müssen wir die restlichen x_i entsprechend ändern, damit weiterhin $\bar{x} = 2$ gilt. Wir könnten das durch $3 \rightarrow 4$ erreichen.
Nun stellt sich die Frage, wie viele der x_i maximal ändern können. Wenn wir alle beliebig ändern, kann es sein, dass $\bar{x} = 2$ nicht mehr gilt. Wenn wir nur $n - 1$ ändern, dann können wir x_n so passend wählen, dass wieder $\bar{x} = 2$ gilt. Also ist der Freiheitsgrad $n - 1$. Da wir $n - 1$ Werte und $\bar{x}$ kennen, können wir den Wert x_n daraus berechnen.

Verteilung von Population und Sample

H_1: Durchschnittliche Länge von Java-Methoden $\mu > 50$
H_0: Durchschnittliche Länge von Java-Methoden $\mu \leq 50$

Gegeben:

- Populations-Verteilung: Kennwerteverteilung der Population P mit Durchschnitt μ und Standardabweichung σ
- Sample-Verteilung: Kennwerteverteilung der Stichproben X mit Durchschnitt $\bar{x}$ und Standardabweichung $s_{\bar{x}}$

Annahmen:

- σ ist bekannt
- P hat Normalverteilung

Daraus folgt: X ist normalverteilt mit $\bar{x} = \mu$ und $s_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$.

Verteilung von Population und Sample

Warum gilt: $\bar{x} = \mu$?

Sample-Größe ist n.
Jeder beobachtete Wert x_i $(1 \leq i \leq n)$ ist eine Messung von einem zufällig ausgewählten Element aus P.
$\rightarrow$ Jede Einzelmessung ist eine Zufallsvariable X_i, deren Verteilung der von P entspricht.

$\bar{x} = \frac{1}{n} \cdot (X_1 + X_2 + \ldots X_n)$

Wenn μ der Durchschnitt von P ist, dann ist μ der Durchschnitt der Verteilung jeder Beobachung X_i.

$\mu_{\bar{x}} = \frac{1}{n} \cdot (\mu_{X_1} + \mu_{X_2} + \ldots \mu_{X_n}) = \frac{1}{n} \cdot (\mu + \mu + \ldots \mu) = \mu$

Verteilung von Population und Sample

Warum gilt: $\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$?

Regeln für Varianzen (a, b sind Konstanten, X, Y Zufallsvariablen):

$$\sigma^2_{a+bX} = b^2\sigma^2_X$$
$$\sigma^2_{X+Y} = \sigma^2_X + \sigma^2_Y$$

Damit:

$\sigma^2_{\bar{x}} = \sigma^2_{\frac{1}{n} \cdot (X_1 + X_2 + \ldots X_n)} = (\frac{1}{n})^2 \cdot (\sigma^2_{X_1} + \sigma^2_{X_2} + \ldots \sigma^2_{X_n})$

Weil jede Einzelbeobachtung X_i aus P stammt, gilt $\sigma^2_{X_i} = \sigma^2$ und damit:

$\sigma^2_{\bar{x}} = (\frac{1}{n})^2 \cdot (\sigma^2 + \sigma^2 + \ldots \sigma^2) = \frac{\sigma^2}{n}$ und $\sigma_{\bar{x}} = \sqrt{\sigma^2_{\bar{x}}} = \frac{\sigma}{\sqrt{n}}$

Verteilung von Population und Sample

H_1: Durchschnittliche Länge von Java-Methoden $\mu > 50$
H_0: Durchschnittliche Länge von Java-Methoden $\mu \leq 50$

Gegeben:

- Populations-Verteilung: Kennwerteverteilung der Population P mit Durchschnitt μ und Standardabweichung σ
- Sample-Verteilung: Kennwerteverteilung der Stichproben X mit Durchschnitt $\bar{x}$ und Standardabweichung $s_{\bar{x}}$

Annahmen:

- σ ist bekannt
- P hat Normalverteilung

Daraus folgt: X ist normalverteilt mit $\bar{x} = \mu$ und $s_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$.

Beispiel

$H_0 : \mu = 50$.

Sei tatsächlich beobachteter Wert (Messung) für $\bar{x} = 54$ mit $\sigma = 10$ und Sample-Größe $n = 25$.

Passt das noch zu H_0 mit Signifikanzniveau $\alpha = 0,01$?

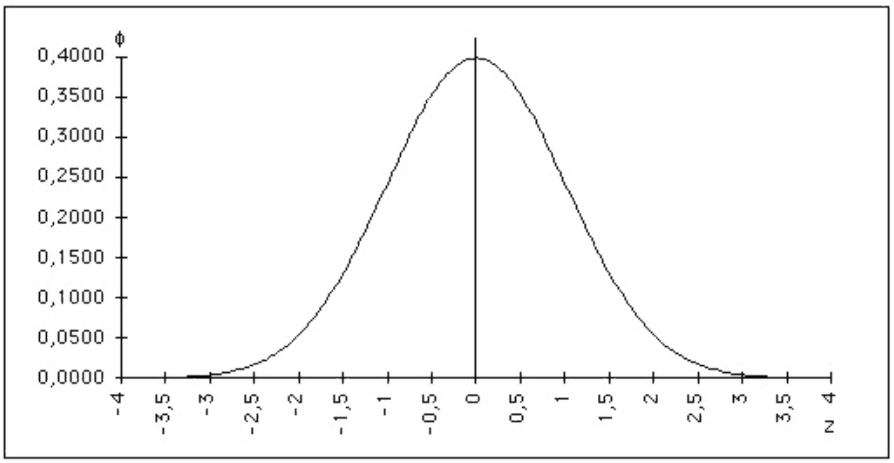
$\bar{x}$ ist normalverteilt mit $\mu = 50$ und $\sigma_{\bar{x}}^2 = \frac{10}{\sqrt{25}} = 2$: $N(50, 2)$

Die Standardnormalverteilung $N(0, 1)$ ist tabelliert. Mit ***z*-Transformation** kann jede Normalverteilung auf $N(0, 1)$ zurückgeführt werden:

$$z_{\bar{x}} = \frac{\bar{x} - \mu}{\sigma_{\bar{x}}}$$

Beispiel

Wahrscheinlichkeit, einen Wert $z_{\bar{x}} = \frac{54-50}{\sqrt{2}} \approx 1,41$ oder größer in $N(0,1)$ zu finden = Flächeninhalt zwischen 1,41 und ∞ in $N(0,1)$



Laut Tabelle für $N(0,1)$: $1 - 0,9207 = 0,0793 > 0,01 = \alpha$.

$\rightarrow H_0$ wird nicht abgelehnt

Wir fragen nach der Wahrscheinlichkeit, mit der Stichprobenergebnisse auftreten können, wenn die Nullhypothese gilt. Wir betrachten nur diejenigen Ergebnisse, die bei Gültigkeit der Nullhypothese höchstens mit einer Wahrscheinlichkeit von α (z.B. 1 % oder 5 %) vorkommen. Gehört das gefundene Stichprobenergebnis zu diesen Ergebnissen, ist das Stichprobenergebnis „praktisch“ nicht mit der Nullhypothese zu vereinbaren. Wir entscheiden uns deshalb dafür, die Nullhypothese abzulehnen und akzeptieren die Alternativhypothese als Erklärung für unser Untersuchungsergebnis.

– Bortz und Döring (2006)

Laut Tabellierung von $N(0,1)$ ist die Fläche von $(-\infty, 1, 41] = 0, 9207$.

Beispieluntersuchung

Hypothese: „Pair-Programming unterscheidet sich in durchschnittlicher Fehlerdichte $\frac{\#Fehler}{LOC}$ von Single-Programming.“

Design:

- Object: Anforderungsspezifikation
- Subjects: 31 professionelle Entwickler
- Blocking:
  - Treatment X: eine Gruppe (10×2) wendet Pair-Programming an
  - Treatment Y: eine Gruppe (11×1) wendet Pair-Programming nicht an

$\rightarrow$ ein Faktor, zwei Treatments

Experiment mit zwei Samples: t-Test

Gegeben: Zwei unabhängige Samples:

$X = x_1, x_2, \ldots x_n$ mit Durchschnitt $\bar{x}$ und Varianz s_x^2
$Y = y_1, y_2, \ldots y_m$ mit Durchschnitt $\bar{y}$ und Varianz s_y^2

H_0: Mittelwerte von X und Y sind gleich: $\mu_x - \mu_y = 0$.

Annahmen:

Population zu X ist normalverteilt mit Durchschnitt μ_x und Varianz σ_x^2,
Population zu Y ist normalverteilt mit Durchschnitt μ_y und Varianz σ_y^2
und $\sigma_x^2 = \sigma_y^2$.

Aber: Varianz σ_x^2 von X und Y ist unbekannt.

Experiment mit zwei Samples: t-Test

Mittelwert von $\bar{x} - \bar{y}$ ist gleich dem Mittelwert von $\mu_x - \mu_y$.

Folgt aus:

- Additionsregel für Mittelwerte
- und Mittelwert von jedem Messwert $\bar{x}$ ist der Mittelwert seiner Population μ

Experiment mit zwei Samples: t-Test

Varianz von $\bar{x} - \bar{y}$ ist:

$$\frac{\sigma_x^2}{n} + \frac{\sigma_y^2}{m}$$

Folgt aus Additionsregel für Varianzen.

Experiment mit zwei Samples: t-Test

Satz: Wenn beide Populationen normalverteilt sind, dann ist die Verteilung von $\bar{x} - \bar{y}$ auch normalverteilt.

z-Transformation einer Zufallsvariablen hat Standardnormalverteilung $N(0, 1)$:

$$z = \frac{(\bar{x} - \bar{y}) - (\mu_x - \mu_y)}{\sqrt{\frac{\sigma_x^2}{n} + \frac{\sigma_y^2}{m}}}$$

Experiment mit zwei Samples: t-Test

Annahme war: beide Populationen haben gleiche Varianz $\sigma_\epsilon^2 = \sigma_x^2 = \sigma_y^2$
Varianz von σ_ϵ^2 kann geschätzt werden durch zusammengelegte Varianzen s_p^2 als gewichteter Durchschnitt:

$$s_p^2 = \frac{(n-1)s_x^2 + (m-1)s_y^2}{(n-1) + (m-1)}$$

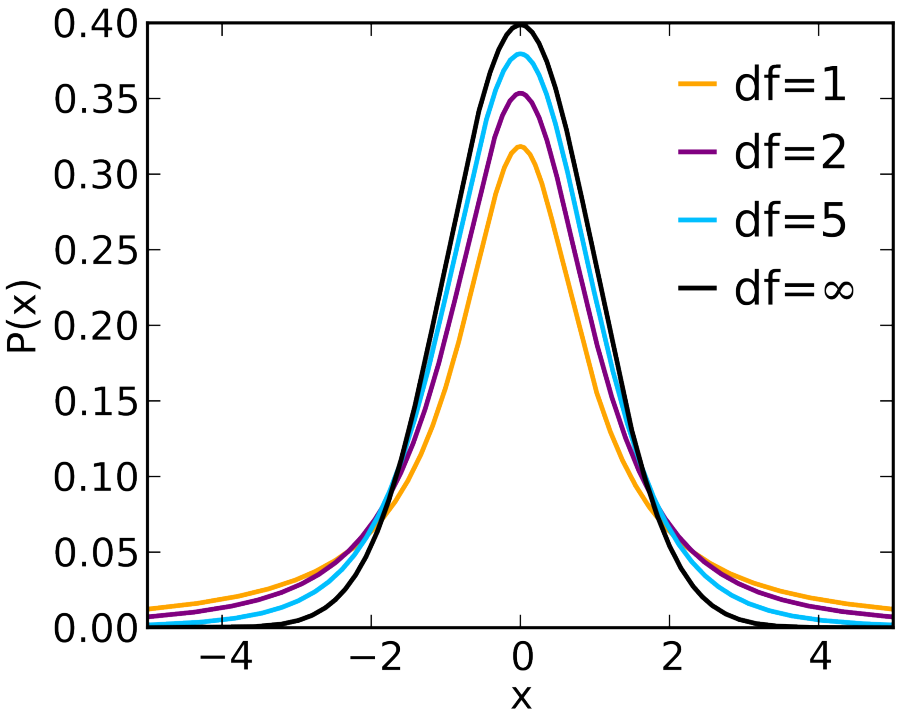
Damit ist z-Transformation für die Schätzung:

$$t = \frac{(\bar{x} - \bar{y}) - (\mu_x - \mu_y)}{\sqrt{\frac{s_p^2}{n} + \frac{s_p^2}{m}}}$$

$\rightarrow$ t folgt Students t-Verteilung mit $(n-1) + (m-1) = n + m - 2$ Freiheitsgraden (df)

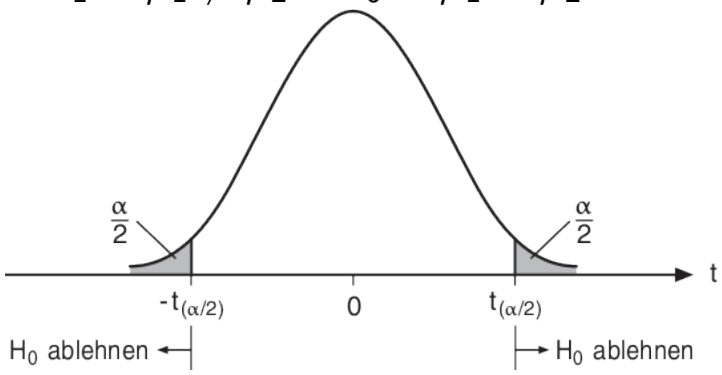
Die Annahme ist, dass die Samples beide eine gemeinsame homogene Varianz haben. Dann kann diese geschätzt werden, indem die Informationen beider Samples gebündelt werden. Die Schätzung ist dann der gewichtete Durchschnitt der einzelnen Varianzen beider Sample-Varianzen. Die Gewichte hierfür sind die jeweiligen Freiheitsgrade $n-1$ und $m-1$. S_p ist dann die gebündelte Varianz.
Der Freiheitsgrad von S_p ist $(n-1)+(m-1)=n+m-2$.

Students t-Verteilung (df = Freiheitsgrad)

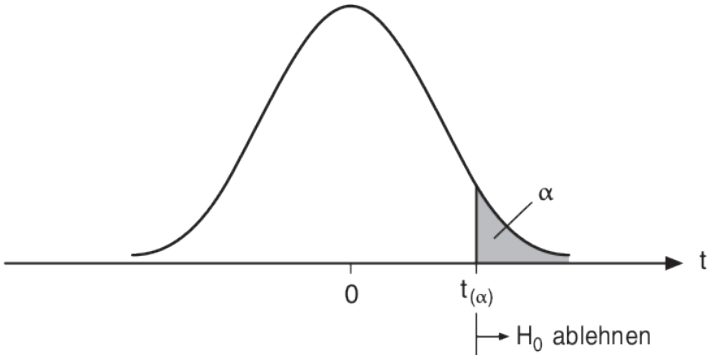


Students t-Verteilung

Ungerichtete $H_1 \equiv \mu_1 \neq \mu_2 \wedge H_0 \equiv \mu_1 = \mu_2 \rightarrow$ zweiseitiger Test



Gerichtete $H_1 \equiv \mu_1 > \mu_2 \wedge H_0 \equiv \mu_1 \leq \mu_2 \rightarrow$ einseitiger Test



Ungerichtete Alternativhypothese $H_1 \equiv \mu_1 \neq \mu_2$**:** Nullhypothese postuliert keinerlei Effekt $H_0 \equiv \mu_1 = \mu_2$.

Gerichtete Alternativhypothese $H_1 \equiv \mu_1 > \mu_2$**:** Nullhypothese postuliert keinen oder gegengerichteten Effekt $H_0 \equiv \mu_1 \leq \mu_2$.

Gerichtete Hypothesen werden anhand der Verteilung über einseitige und ungerichtete Hypothesen über zweiseitige Tests geprüft. Bei einem zweiseitigen Test markieren die Werte t(α/2) und -t(α/2) diejenigen t-Werte einer t-Verteilung, die von den Extremen der Verteilungsfläche jeweils $\alpha/2\,\%$ abschneiden.

Zusammenfassung des Vorgehens beim t-Test

- Eingabe: Zwei unabhängige Samples $x_1, x_2, \ldots x_n$ und $y_1, y_2 \ldots y_m$
- Annahme: Populationen zu X und Y sind normalverteilt und haben gleiche Varianz
- H_0: Mittelwerte von X und Y sind gleich: $\mu_x - \mu_y = 0$
- Transformation von H_0: $t_0 = \frac{\bar{x}-\bar{y}}{s_p \times \sqrt{\frac{1}{n}+\frac{1}{m}}}$

  wobei $s_p = \sqrt{\frac{(n-1)s_x^2+(m-1)s_y^2}{(n-1)+(m-1)}}$

  und s_x^2 und s_y^2 sind die individuellen Sample-Varianzen
- t_0 folgt bei Gültigkeit von H_0 einer t-Verteilung mit $n+m-2$ Freiheitsgraden
- Kriterium (mit Signifikanzniveau α); H_0 ablehnen, wenn
  - $|t_0| > t_{\alpha/2,n+m-2}$ (ungerichtete Hypothese)
  - $|t_0| > t_{\alpha,n+m-2}$ (gerichtete Hypothese)

Beispielmessungen

Treatment X = Pair-Programming, Treatment Y = kein Pair-Programming

i	Treatment X: x_i	Treatment Y: y_i
1	3,24	3,44
2	2,71	4,97
3	2,84	4,76
4	1,85	4,96
5	3,22	4,10
6	3,48	3,05
7	2,68	4,09
8	4,30	3,69
9	2,49	4,21
10	1,54	4,40
11		3,49
	n=10	m=11
	$\bar{x} = 2,835$	$\bar{y} = 4,1055$
	$S_x^2 = 0,6312$	$S_y^2 = 0,4112$

Beispielauswertung mit t-Test

$$\begin{aligned} s_p &= \sqrt{\frac{(n-1)s_x^2+(m-1)s_y^2}{(n-1)+(m-1)}} \\ &= \sqrt{\frac{(10-1)\cdot 0{,}6312+(11-1)\cdot 0{,}4112}{(10-1)+(11-1)}} = -0,564 \end{aligned}$$

$$\begin{aligned} t_0 &= \frac{\bar{x}-\bar{y}}{s_p \times \sqrt{\frac{1}{n}+\frac{1}{m}}} \\ &= \frac{2{,}835-4{,}1055}{-0{,}564\times\sqrt{\frac{1}{10}+\frac{1}{11}}} = -5,642 \end{aligned}$$

Freiheitsgrade: $df = 10 + 11 - 2 = 19$
$\rightarrow t_{\alpha/2,n+m-2} = t_{0{,}05/2,10+11-2} = 2,093$

$|t_0| = 5,642 > t_{0{,}05/2,10+11-2} = 2,093 \rightarrow H_0$ ablehnen

Siehe z.B. http://www.math.unb.ca/~knight/utility/t-table.htm für eine Tabelle der Students t-Verteilung.

Exakter U-Test von Mann-Whitney

Gegeben: zwei unabhängige Samples $x_1, x_2, \ldots x_n$ und $y_1, y_2, \ldots y_m$ mit Ordinalskalenniveau.

Annahme: Beide Samples stammen von Populationen mit der gleichen Verteilung.

Keine Annahme über diese Verteilung.

1. Daten beider Samples werden vereinigt und geordnet.
2. Jeder Wert x_i wird mit jedem Wert y_j verglichen:
   - G_i = Anzahl der $y_j < x_i$
   - L_i = Anzahl der $y_j > x_i$
3. Summiere:
   - $G = \sum_{1 \leq i \leq n} G_i$
   - $L = \sum_{1 \leq i \leq n} L_i$
   - $U = \min(L, G)$

Gruppe	x_i bzw. y_i	G_i	L_i
X	1.54	11	0
X	1.85	11	0
X	2.49	11	0
X	2.68	11	0
X	2.71	11	0
X	2.84	11	0
Y	3.05		
X	3.22	10	1
X	3.24	10	1
Y	3.44		
X	3.48	9	2
Y	3.49		
Y	3.69		
Y	4.09		
Y	4.10		
Y	4.21		
X	4.30	4	7
Y	4.40		
Y	4.76		
Y	4.96		
Y	4.97		
	$\sum$	99	11

Signifikanztest zum exakten U-Test von Mann-Whitney

- Es gibt $\binom{n+m}{m} = \binom{n+m}{n}$ mögliche Rangfolgen.
- Erwartungswert für U bei H_o: $\mu_U = (n+m)/2$.
- Je weiter beobachtetes U vom Erwartungswert abweicht, desto unwahrscheinlicher ist H_0.
- Einseitiger Test:
  - Z_U = Anzahl möglicher Kombinationen, die einen U-Wert liefern, der nicht größer als U ist.
  - $P = Z_U / \binom{n+m}{m}$
- Zweiseitiger Test:
  - Z'_U = Anzahl möglicher Kombinationen, die einen U-Wert liefern, der nicht kleiner als $max(L, G)$ ist.
  - $P = (Z_U + Z'_U) / \binom{n+m}{m}$
- Lehne H_0 ab, wenn $P \leq \alpha$.
- Kritischer Wert (der zur Ablehnung von H_0 führt) kann in Tabelle des U-Tests für kleine Samples nachgeschlagen werden.
  Im Beispiel: kritischer Wert $= 26$ für $\alpha = 0,05$
  $\rightarrow$ H_0 wird abgelehnt wegen $U < 26$

Tabellen für den kritischen Wert bei gegebenem Signifikanzniveau für den U-Test lassen sich im Web finden, indem man nach den Stichwörtern „table u test“ sucht. Z.B.: `math.usask.ca/~laverty/S245/Tables/wmw.pdf`
Es wird vorausgesetzt, dass keine identischen Messwerte („Bindungen“oder „Rangbindungen“) auftreten. Falls Bindungen vorhanden sind, werden den Werten die mittleren Rangzahlen zugewiesen.

Weiterführende Literatur

Empirische Methoden

- Endres und Rombach (2003) beschreiben wesentliche empirische Kenntnisse in der Software-Technik und brechen eine Lanze für die empirische Forschung in diesem Gebiet.
- Lienert (1973) beschreibt verteilungsfreie (nicht-parametrische) statistische Tests
- Prechelt (2001) beschreibt empirische Methoden in der Softwaretechnik (deutschsprachig, leider vergriffen und wird nicht mehr neu aufgelegt)
- Wohlin u. a. (2000) beschreibt empirische Methoden in der Softwaretechnik
- Christensen (2007) beschreibt experimentelle Methoden im Allgemeinen

Weiterführende Literatur

Statistik in der Empirie

- Bortz u. a. (2008) beschreiben experimentelle Designs und ihre statistischen (nicht-parametrischen, d.h. verteilungsfreien) Auswertungen
- Winner u. a. (1991) beschreiben experimentelle Designs und ihre statistischen (parametrischen) Auswertungen
- Moore u. a. (2009) geben eine allgemeine Einführung in Statistik

Wiederholungs- und Vertiefungsfragen

- Was ist ein statistische Hypothese? Wie wird sie überprüft und welche Rolle spielt dabei das Signifikanzniveau (der kritische Wert)?
- Welche Arten von Hypothesen gibt es?
- Mit welchen Maßen werden Population und Sample meist statistisch charakterisiert?
- Was versteht man unter einem parametrischen bzw. nichtparametrischen Test?
- Erläutern Sie das Prinzip des t-Tests.
- Erläutern Sie das Prinzip des exakten U-Tests von Mann-Whitney.

Entwurfsmuster

Entwurfsmuster

- Kategorien von Entwurfsmustern
- Bestandteile eines Entwurfsmusters
- Entwurfsmuster Adapter
- Entwurfsmuster Command
- Entwurfsmuster Decorator
- Entwurfsmuster Strategy
- Entwurfsmuster für Synchronisation: Scoped Locking
- Entwurfsmuster für Synchronisation: Strategized Locking
- Entwurfsmuster für Synchronisation: Thread-Safe Interface
- Entwurfsmuster für Parallelisierung: Thread Pool
- Entwurfsmuster für Parallelisierung: Monitor Object
- Entwurfsmuster für Parallelisierung: Leader/Followers
- Wiederholungsfragen

Lernziele

- Verstehen, was Entwurfsmuster sind
- Verschiedene Enwurfsmuster kennen und anwenden können
- Qualitäten und Einsetzbarkeit der Entwurfsmuster kennen

Entwurfsmuster

> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*
>
> Christopher Alexander (Architekt und Mathematiker), "A pattern language", 1977.

Definition

Entwurfsmuster: "Musterlösung" für ein wiederkehrendes Entwurfsproblem.

Kategorien von Entwurfsmustern

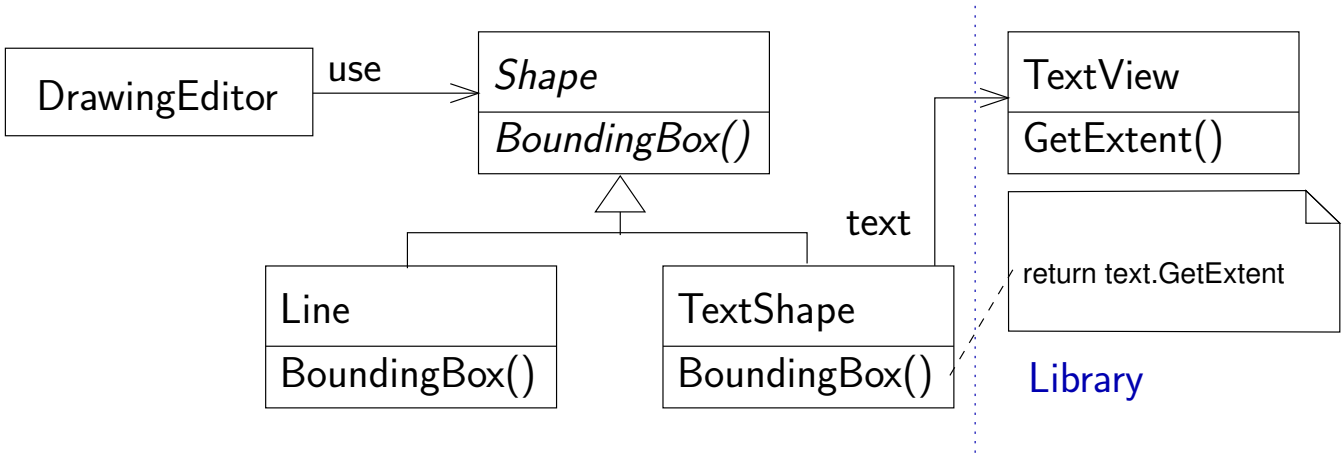
- Muster für das Erzeugen von Instanzen
  - Singleton
- strukturelle Muster zur Komposition von Klassen oder Objekten
  - Composite
  - Adapter
  - Decorator
- Verhaltensmuster betreffen Interaktion von Klassen oder Objekten
  - Command

Bestandteile eines Entwurfsmusters

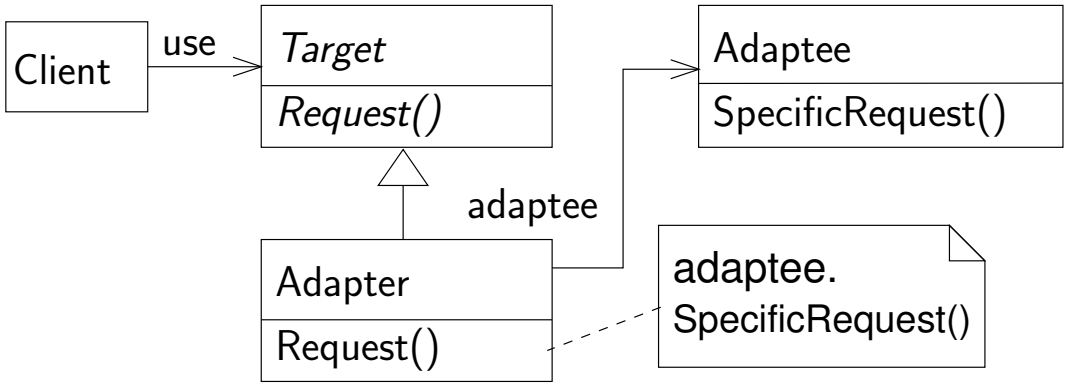
- **Name** (kurz und beschreibend)
- **Problem**: Was das Entwurfsmuster löst
- **Lösung**: Wie es das Problem löst
- **Konsequenzen**: Folgen und Kompromisse des Musters.

Problem

- eine vorhandene wiederverwendbare Komponente hat nicht die passende Schnittstelle,
- und der Code der Komponente kann nicht verändert werden



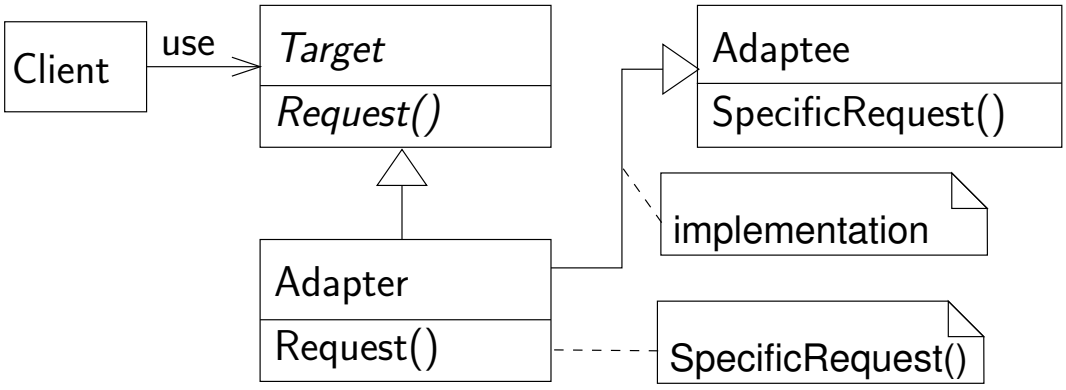
Lösung: Objekt-Adapter



Konsequenzen:

- erlaubt Anpassung von Adaptee und all seiner Unterklassen auf einmal
- Overriding von Adaptees Methoden schwierig
  - → Ableitung von Adaptee
  - → Adapter referenziert auf Ableitung
- führt Indirektion ein

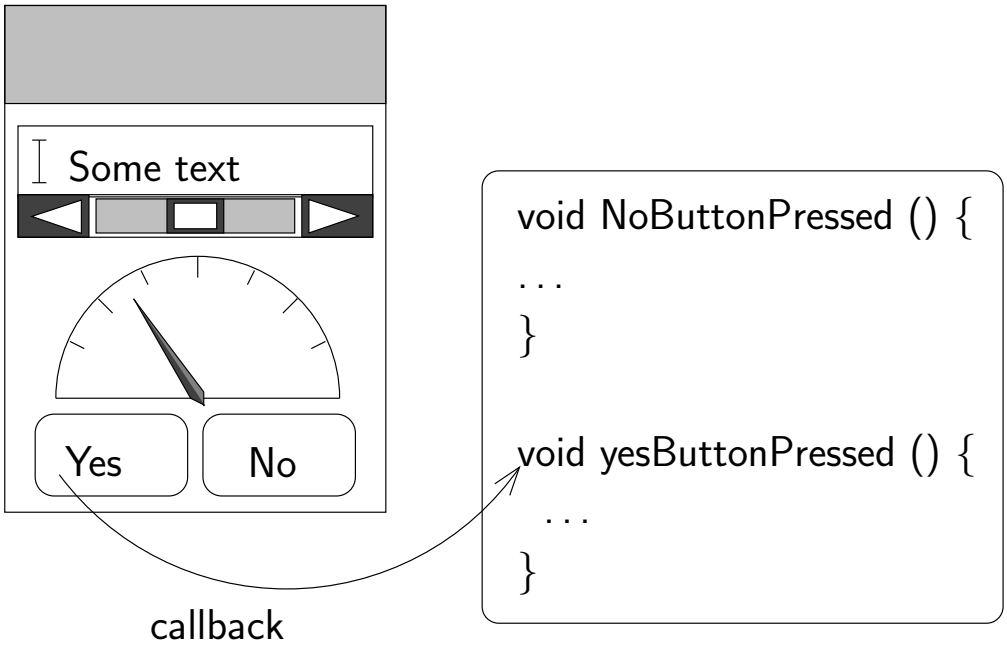
Lösung: Klassen-Adapter



Konsequenzen:

- keine Indirektion
- setzt Mehrfachvererbung voraus
- passt nur Adaptee an, nicht seine Unterklassen
- Overriding von Adaptees Methoden einfach

Problem



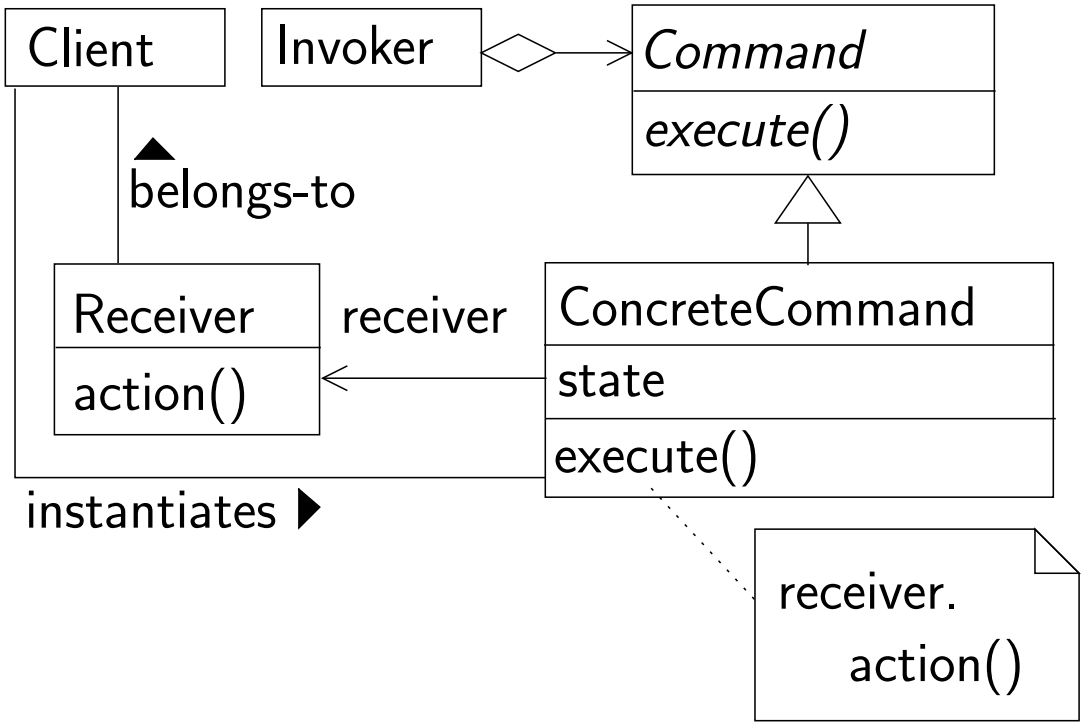
Lösung in C: Klient

```
1  void YesButtonPressed () {...}
2  void NoButtonPressed ()  {...}
3
4  Button mybutton;
5
6  int main (){
7    attach (&mybutton, &YesButtonPressed);
8    ...
9    event_loop ();
10 }
```

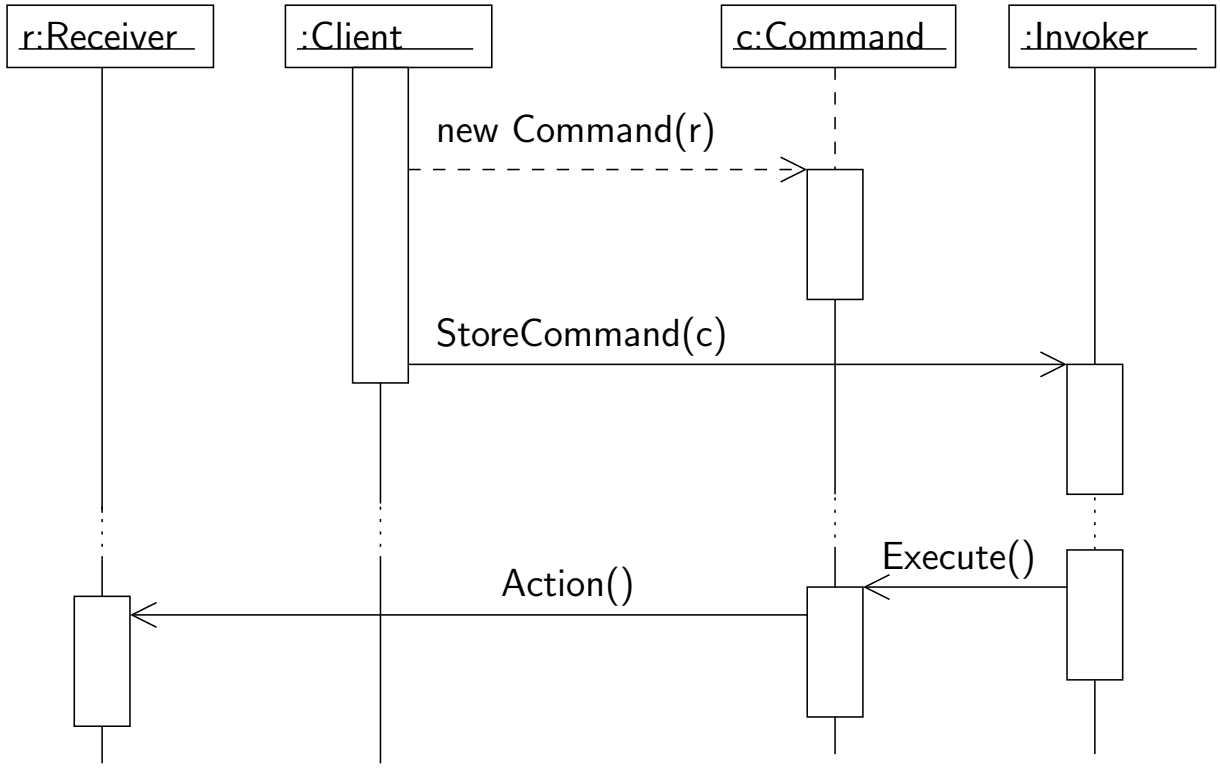
Lösung in C: Mechanismus

```c
typedef void (*Callback)();

typedef struct Button {
  Callback execute;
  int x;
  int y;
} Button;

void attach (Button *b, Callback execute) {
  b->execute = execute;
}

void event_loop (){
  ...
  widgets[i]->execute();
  ...
}
```


Lösung mit OOP



Lösung mit OOP

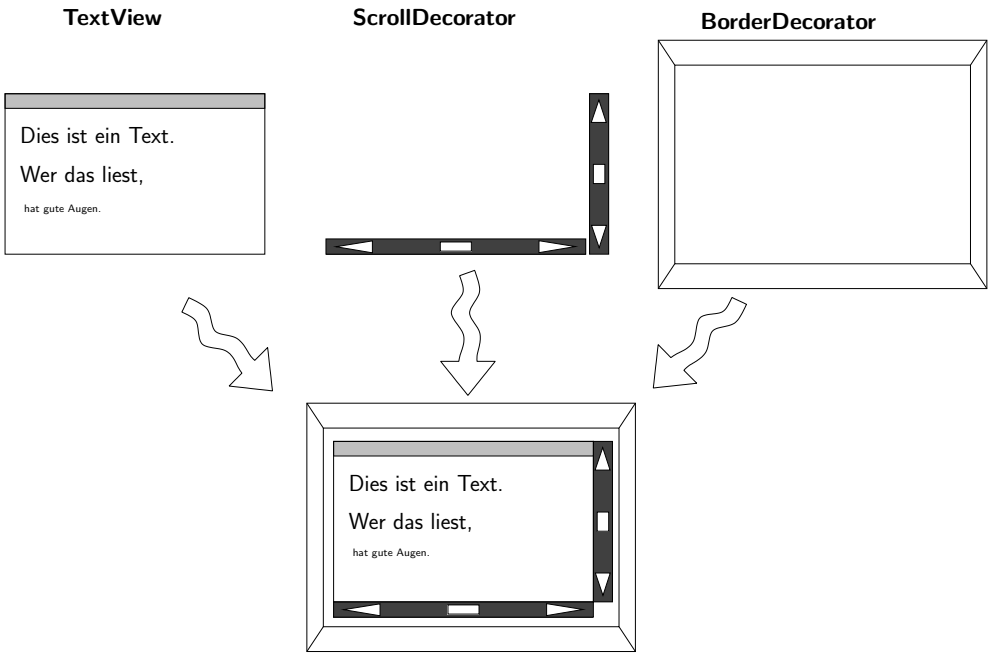


Konsequenzen

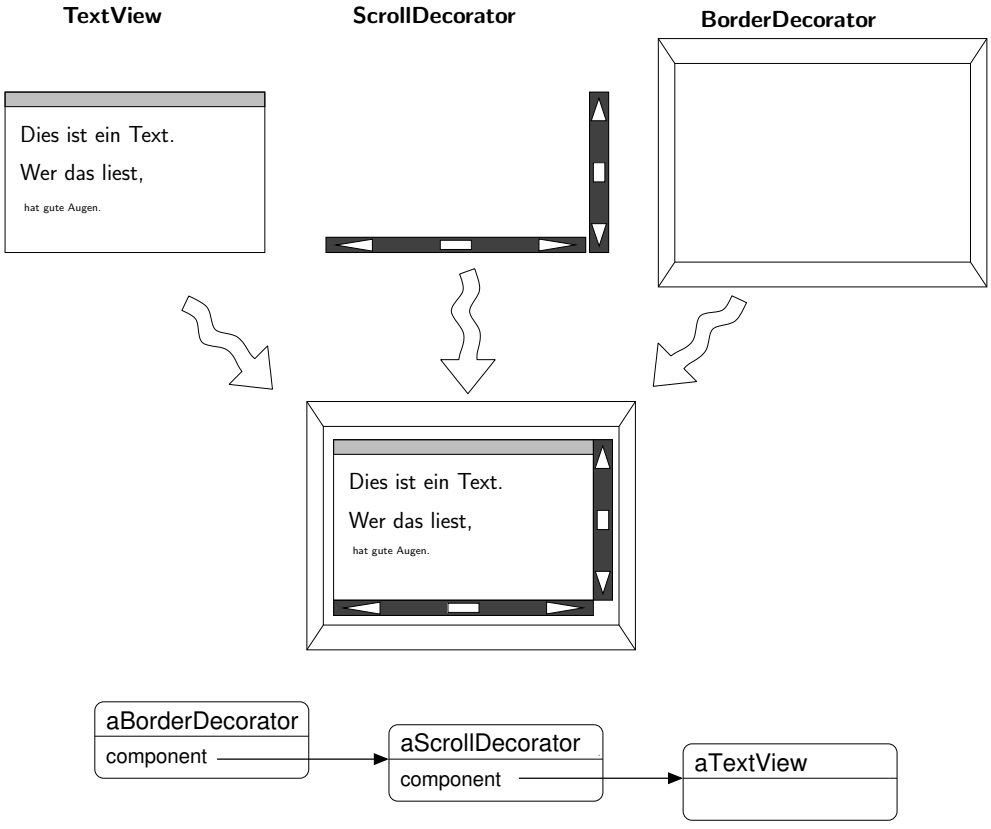
- Entkopplung von Objekt, das Operation aufruft, von dem, welches weiß, wie man es ausführt
- Kommandos sind selbst Objekte und können als solche verwendet werden (Attribute, Vererbung etc.)
- Hinzufügen weiterer Kommandos ist einfach

Problem

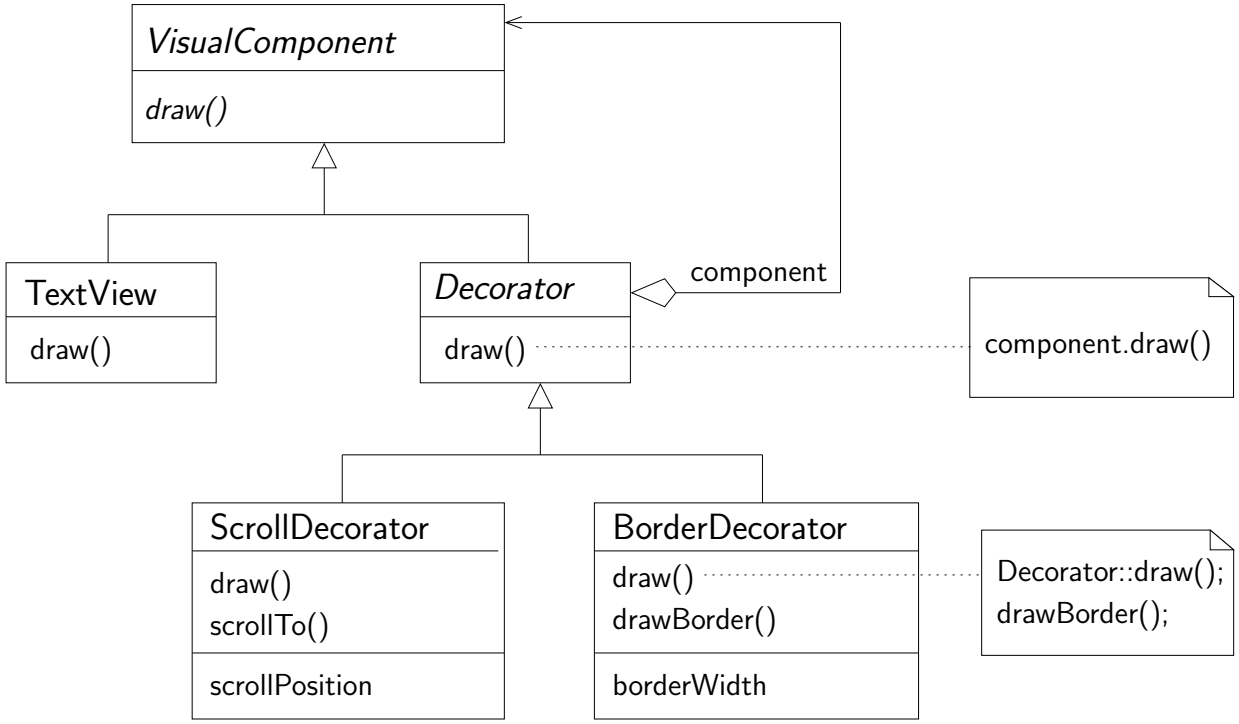
- Eigenschaften sollen einzelnen Objekten, nicht ganzen Klassen, zugewiesen werden können
- Zuweisung soll dynamisch geschehen



Problem



Lösung



Konsequenzen

- höhere Flexibilität als statische Vererbung
- vermeidet Eier-Legende-Wollmilchsau-Klassen
- Dekorator und dekoriertes Objekt sind nicht identisch
- vielfältige dynamische Kombinationsmöglichkeiten
  $\rightarrow$ schwer statisch zu analysieren

Entwurfsmuster Strategy (prozedural)

```
1  typedef enum Strategy {s1, s2, s3} Strategy;
2
3  void apply (Strategy s) {
4
5    switch (s) {
6      case s1:  applyS1(); break;
7      case s2:  applyS2(); break;
8      case s3:  applyS3(); break;
9    }
10
11 }
```


Entwurfsmuster Strategy mit OO-Techniken

```
1  class Applier {
2  public:
3   Strategy *strategy
4
5   void apply () {
6     strategy->algorithm ();
7   }
8  }
9
10 class Strategy {
11   virtual void algorithm () = 0;
12 }
13
14 class Strategy1 : Strategy {
15   virtual void algorithm () {...}
16 }
17 ...
18 Applier a;
19 a.strategy = StrategyFactory.instance();
20 a.apply();
```

Mutex

```
#ifndef MUTEX_H
#define MUTEX_H
#include"Lock.h"
#include <pthread.h>
class Mutex : Lock
{
 public:
   Mutex();            // Creates the Mutex.
   virtual ~Mutex();   // Destroys the Mutex.
   void lock();        // Locks the mutex. Blocks if the mutex
                       // is held by another thread.
   void unlock();      // Unlocks the mutex so that it can be
                       // acquired by other threads.
 private:
   pthread_mutex_t mutex;
   friend class Condition;
};
#endif
```

Locking mit Mutex

```
1 #include"Mutex.h"
2 class Buffer {
3 public:
4   Buffer() : index(-1) {};
5   bool insert(int i) {
6     mutex.lock(); // critcial section
7     if(index >= 100) {
8       mutex.unlock();
9       return false;
10    }
11    index++;
12    content[index] = i;
13    mutex.unlock();
14    return true;
15  };
16 private:
17  Mutex mutex;
18  int content[100];
19  int index;
20 };
```


Fragen

Wie kann man sich davon befreien, sich explizit um das Locking und Unlocking kümmern zu müssen?

Scoped Locking nach Schmidt u. a. (2000)

```
1  #include"Mutex_Guard.h"
2  class Buffer {
3  public:
4    Buffer() : index(-1) {};
5    bool insert(int i) {
6      // use scoped locking to acquire and release
7      // lock automatically
8      Mutex_Guard guard(mutex);
9      if(index >= 100) {
10       return false;
11     }
12     index++;
13     content[index] = i;
14     return true;
15   };
16 private:
17   Mutex mutex;
18   int content[100];
19   int index;
20 };
```


> Das C++ Idiom *Scoped Locking* stellt sicher, dass ein Lock erlangt wird, wenn die Kontrolle einen kritischen Abschnitt betritt, und der Lock wieder frei gegeben wird, wenn der Abschnitt wieder verlassen wird, völlig unabhängig davon, auf welchem Kontrollflusspfad dies geschieht.
>
> – Schmidt u. a. (2000)

Scoped Locking nach Schmidt u. a. (2000)

```
1  #ifndef MUTEX_GUARD_H
2  #define MUTEX_GUARD_H
3  #include "Mutex.h"
4  class Mutex_Guard {
5   public:
6    // store a pointer to the mutex m and acquire it
7    Mutex_Guard(Mutex &m) : mutex(&m), owner(false) {
8      mutex->lock();
9      owner = true;  // only true iff lock() succeeds
10   }
11   // release the mutex when guard leaves the scope
12   ~Mutex_Guard() {
13     // only release mutex if it was acquired successfully
14     if(owner) mutex->unlock();
15   }
16  private:
17   Mutex *mutex;
18   bool owner;
19   // disallow copying and assignment
20   Mutex_Guard(const Mutex_Guard&);
21   void operator=(const Mutex_Guard&);
22 };
23 #endif
```


Vorteil: Keine explizite Behandlung von lock und unlock, da impliziter Sprachmechanismus ausgenutzt wird.

Das C++ Idiom *Scoped Locking* stellt sicher, dass ein Lock erlangt wird, wenn die Kontrolle einen kritischen Abschnitt betritt, und der Lock wieder frei gegeben wird, wenn der Abschnitt wieder verlassen wird, völlig unabhängig davon, auf welchem Kontrollflusspfad dies geschieht.

– Schmidt u. a. (2000)

Fragen

Wie kann man den Locking-Mechanismus parameterisieren?

Strategized Locking nach Schmidt u. a. (2000)

```
1  #include "Guard.h"
2  class Buffer {
3  public:
4    Buffer(Lock &l) : lock(&l), index(-1) {};
5    bool insert(int i) {
6      Guard guard(*lock);
7      // critical section
8      // ...
9      return true;
10   };
11 private:
12   Lock *lock;
13   int content[100];
14   int index;
15 };
```


Lock soll verschiedene Implementierungen haben können.
Im Konstruktor von *Buffer* kann die konkrete Implementierung übergeben werden.

Strategized Locking nach Schmidt u. a. (2000)

```
1  #include"Lock.h"
2  class Guard {
3    public:
4     // store a pointer to the mutex m and acquire it
5     Guard(Lock &m) : lock(&m), owner(false) {
6       lock->lock();
7       owner = true;  // only true iff lock() succeeds
8     }
9     // release the lock when guard leaves the scope
10    ~Guard() {
11      // only release lock if it was acquired successfully
12      if(owner) lock->unlock();
13    }
14   private:
15    Lock *lock;
16    bool owner;
17    // disallow copying and assignment
18    Guard(const Guard&);
19    void operator=(const Guard&);
20 };
```


> Lock soll verschiedene Implementierungen haben können.
> Das Entwurfsmuster *Strategized Locking* parameterisiert einen Synchronisationsmechanismus, mit dessen Hilfe die kritischen Abschnitte vor paralleler Ausführung geschützt werden.
>
> – Schmidt u. a. (2000)

Strategized Locking nach Schmidt u. a. (2000)

```
1 #ifndef LOCK_H
2 #define LOCK_H
3 class Lock
4 {
5   public:
6     virtual void lock() = 0;
7       // Acquires the lock. Blocks if the lock
8     // is held by another thread.
9     virtual void unlock() = 0;
10    // Releases the lock so that it can be
11    // acquired by other threads.
12 };
13 #endif
```


`Lock` ist abstrakte Klasse (Schnittstelle).

Strategized Locking nach Schmidt u. a. (2000)

```
1 #include"Lock.h"
2
3 class NullableLock : Lock
4 {
5  public:
6    NullableLock();
7    ~NullableLock();
8    inline void lock() {}
9    inline void unlock() {}
10 };
```


`Mutex` implementiert `Lock`. Falls es keine Threads gibt, dann kann man `Nullable` verwenden.
Die Entscheidung kann hier zur Laufzeit getroffen werden.
Falls das bereits statisch bekannt ist und man den Laufzeit-Overhead vermeiden möchte, dann kann man Templates benutzen.

Strategized Locking nach Schmidt u. a. (2000)

```
#include"Lock.h"
template <class LOCK>
class Guard_Template {
 public:
  // store a pointer to the mutex m and acquire it
  Guard_Template(LOCK &m) : lock(&m), owner(false) {
    lock->lock();
    owner = true;  // only true iff lock() succeeds
  }
  // release the lock when guard leaves the scope
  ~Guard_Template() {
    // only release lock if it was acquired successfully
    if(owner) lock->unlock();
  }
 private:
  LOCK *lock;
  bool owner;
  // disallow copying and assignment
  Guard_Template(const Guard_Template&);
  void operator=(const Guard_Template&);
};
```


Strategized Locking nach Schmidt u. a. (2000)

```
#include"Guard_Template.h"
template <class LOCK>
class Buffer {
public:
  Buffer() : index(-1) {};
  bool insert(int i) {
    Guard_Template<LOCK> guard(lock);
    // critical section
    // ...
    return true;
  };
private:
  LOCK lock;
  int content[100];
  int index;
};
```

Der `Buffer` wird selbst zum Template und kann auf diese Weise statisch parametrisiert werden.

Fragen

Wie vermeidet man unnötiges Locking und stellt sicher, dass Aufrufe eigener Methoden nicht zu Deadlocks führen?

Deadlock

```
#include"Guard.h"
class Buffer {
public:
  Buffer(Lock &l) : lock(&l), index(-1) {};
  bool insert(int i) {
    Guard guard(*lock);
    if (size() == 100) return false;
    else {
      // ....
      return true;
    }
  };
  const int size() {
    Guard guard(*lock);
    return index + 1;
  }
private:
  Lock *lock;
  int content[100];
  int index;
};
```


Thread-Safe Interface nach Schmidt u. a. (2000)

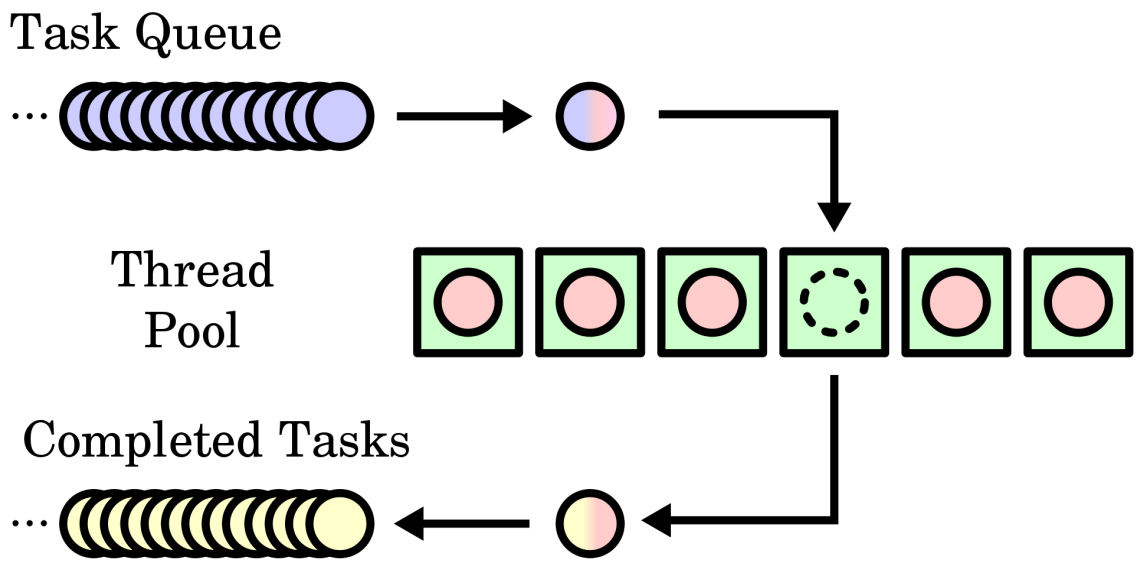
```
#include"Guard.h"
class Buffer {
public:
  Buffer(Lock &l) : lock(&l), index(-1) {};
  bool insert(int i) {
    Guard guard(*lock);
    return insert_unlocked(i);
  };
  const int size() {
    Guard guard(*lock);
    return size_unlocked();
  }
private:
  Lock *lock;
  int content[100];
  int index;
  bool insert_unlocked(int i) {
    if (size_unlocked() == 100) return false;
    // ....
  }
  const int size_unlocked() { return index + 1; }
};
```

Das Entwurfsmuster *Thread-Safe Interface* vermeidet unnötiges Locking und Deadlocks durch Aufrufe eigener Methoden, indem Synchronisation am Interface von der Implementierung getrennt wird.

Definition

Embarrasingly parallel problem: Ein Problem, das ohne Mühe in parallel abarbeitbare Aufgaben zerlegt werden kann, zwischen denen keine Abhängigkeiten existieren.

Thread Pool Pattern (auch: *Replicated Workers*)



A simple thread pool. The task queue has many waiting tasks (blue circles). When a thread opens up in the queue (green box with dotted circle) a task comes off the queue and the open thread executes it (red circles in green boxes). The completed task then "leaves" the thread pool and joins the completed tasks list (yellow circles).
Author: Colin M.L. Burnett
Permission under license: GFDL

Thread Pool Pattern

```
1 class Task {
2  public:
3   void solve();
4   void define(int i);
5  private:
6   int data;
7 };
```

Thread Pool Pattern

```
#include"Mutex_Guard.h"
#include"Task.h"
#include <vector>
#include <exception>
using namespace std;
class Queue {
public:
  void put(Task t) { // put t in queue
    Mutex_Guard guard(mutex);
    tasks.push_back(t);
  }
  Task get() { // retrieve task from queue
    Mutex_Guard guard(mutex);
    if(tasks.size()==0) throw exception();
    Task result = tasks.back();
    tasks.pop_back();
    return result;
  }
private:
  Mutex mutex;
  vector<Task> tasks;
};
```


Thread Pool Pattern

```
Queue queue;

// entry point of thread
void *worker(void *ptr)
{
  while(true) {
      try
      { Task t = queue.get();
        t.solve();
      }
      catch(exception &e)
      { break;}
  }
}
```

Thread Pool Pattern

```
1  main()
2  {
3    // define work
4    for(int i = 0; i < 100; i++) {
5      Task *t = new Task();
6      t->define(i);
7      queue.put(*t);
8    }
9
10   std::vector<pthread_t> thread_pool(10);
11
12   // Create independent threads each of which
13   // will execute function worker.
14   for(int i = 0; i < thread_pool.size(); i++)
15     pthread_create(&thread_pool[i], NULL, worker, NULL);
16
17   // Wait till threads are complete before main continues.
18   for(int i = 0; i < thread_pool.size(); i++)
19     pthread_join(thread_pool[i], NULL);
20 }
```


Fragen

Was tun, wenn es Produzenten gibt, die kontinuierlich Aufgaben erzeugen?

Beliebig viele Produzenten und Konsumenten

```
1  Queue_Monitor queue(5);
2
3  // entry point of consumer thread
4  void *consumer(void *ptr)
5  {
6    while(true)
7    { Task t = queue.get();
8      t.solve();
9    }
10 }
11
12 // entry point of producer thread
13 void *producer(void *ptr)
14 { Task t;
15   int data = *(int*)ptr;
16   while(true)
17   { t.define(data);
18     queue.put(t);
19     wait(2);
20   }
21 }
```


Monitor Object

```
1  #include "Mutex_Guard.h"
2  #include "Task.h"
3  #include "Condition.h"
4  class Queue_Monitor {
5  public:
6    Queue_Monitor(int capacity);
7    ~Queue_Monitor();
8    Task get();
9    // blocks if no task available
10   void put(Task t);
11   // blocks if no space left
12 private:
13   Mutex mutex;          // monitor lock
14   Condition not_full;   // wait for queue no longer full
15   Condition not_empty; // wait for queue no longer empty
16   Task *tasks;
17   int size, max_size;
18 };
```

Monitor Condition

```
#ifndef CONDITION_H
#define CONDITION_H
#include <pthread.h>
#include "Mutex.h"
class Condition
{
 public:
   Condition(Mutex &m); // Creates the Condition.
   ~Condition();        // Destroys the Condition.
   void wait();         // Puts executing thread to sleep
   void notifyAll(); // Wakes up all sleeping threads.
 private:
   pthread_cond_t condition;
   Mutex &mutex;
};
#endif
```


Monitor Object I

```
#include "Monitor.h"

Queue_Monitor::Queue_Monitor(int capacity)
    : size(0), max_size(capacity),
      not_full(mutex), not_empty(mutex)
{ tasks = new Task[capacity]; }

Queue_Monitor::~Queue_Monitor()
{ delete [] tasks; }

Task Queue_Monitor::get() {
  Mutex_Guard guard(mutex);
  while(size==0) {
    not_empty.wait();
  }
  not_full.notifyAll();
  return tasks[--size];
}

void Queue_Monitor::put(Task t) {
```

Monitor Object II

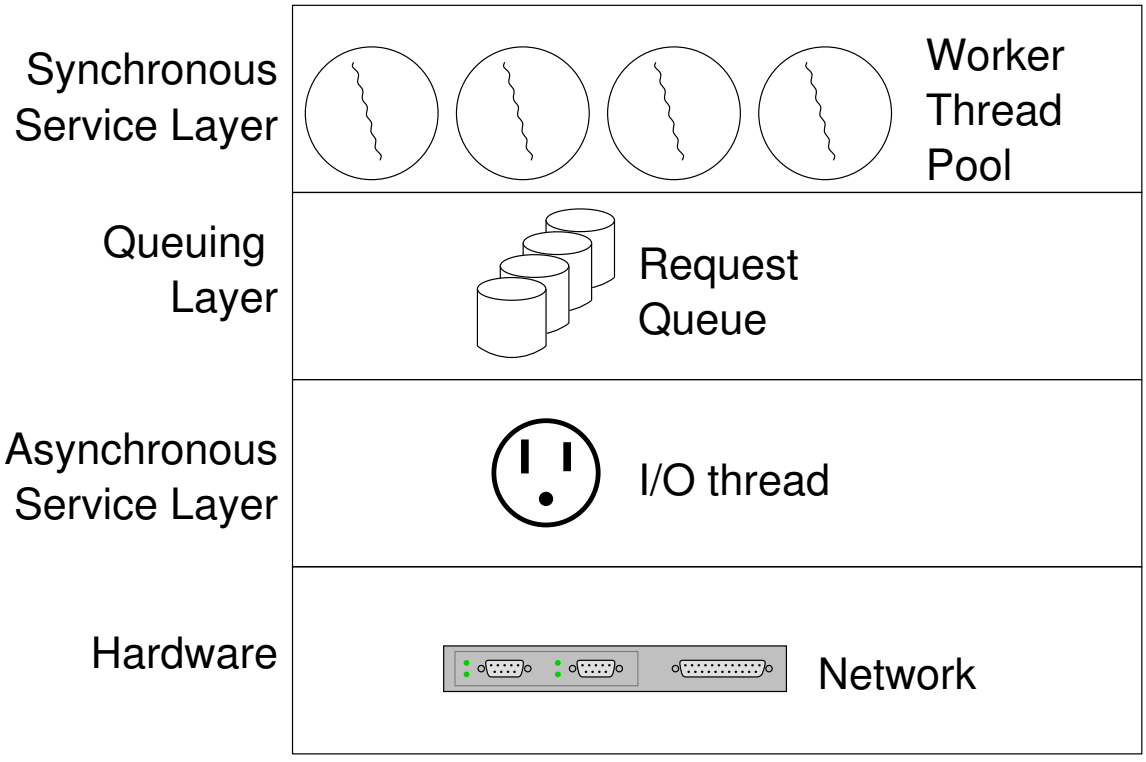
```
21    Mutex_Guard guard(mutex);
22    while(size==max_size) {
23      not_full.wait();
24    }
25    not_empty.notifyAll();
26    tasks[size++] = t;
27  }
```


Client-Code

```
1  while(true){
2    // creating a stream (TCP) socket with Poco; it is bound to
3    // a local source port and a network interface address
4    // (hostname, port)
5    Handle socket(Poco::Net::SocketAddress(hostname, port));
6    // send request
7    write(socket, message_type() + toString(i));
8    // get answer
9    std::cout << "reply: " << read(socket) << std::endl;
10   sleep(1);
11   i++;
12 }
```

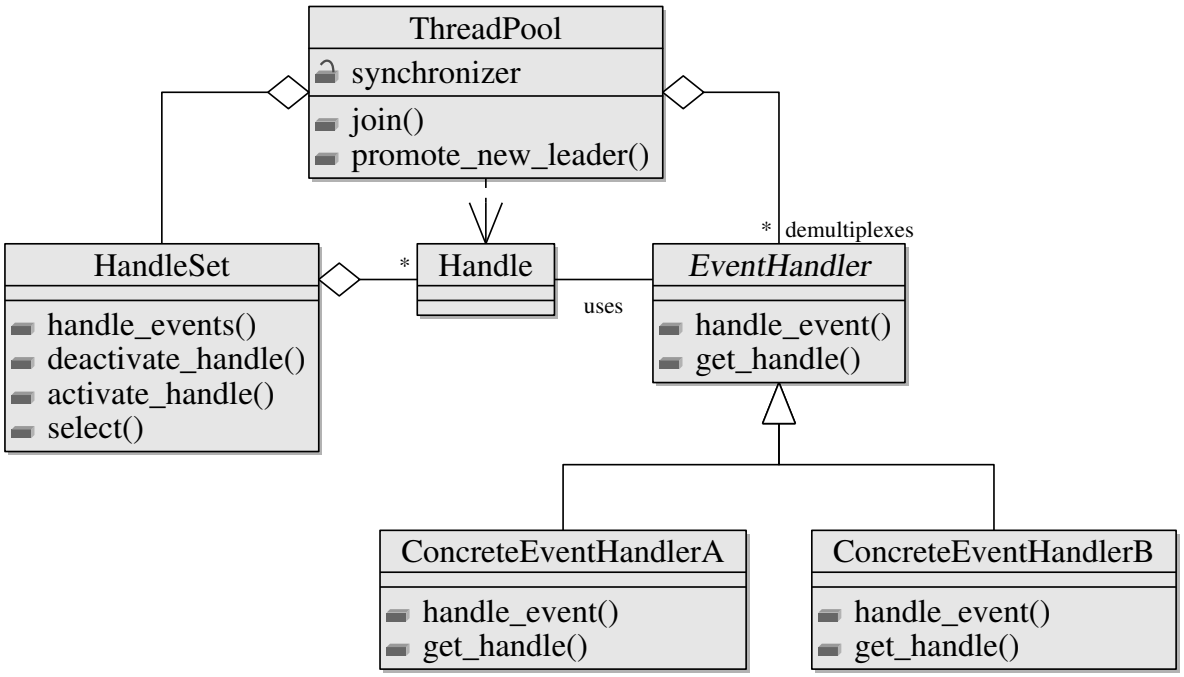
Server-Code

```
while(true){
    // waits for any request at all readSockets
    int ready_sockets
        = Poco::Net::Socket::select(readSockets, writeSockets,
                                    exceptSockets, 10000000);
    // readSockets now contains all sockets where requests are
    // available; ready_sockets tells how many sockets exist
    // for which we have requests
    if (ready_sockets > 0) {
      // we handle only one of the requests in readSockets;
      // the others will be served in the next loop iteration
      Handle connection =
          ((Poco::Net::ServerSocket)readSockets[0])
              .acceptConnection();
      // handle the request by delegating to an EventHandler
      EventHandler handler(connection);
      handler.handle_event();
    }
}
```

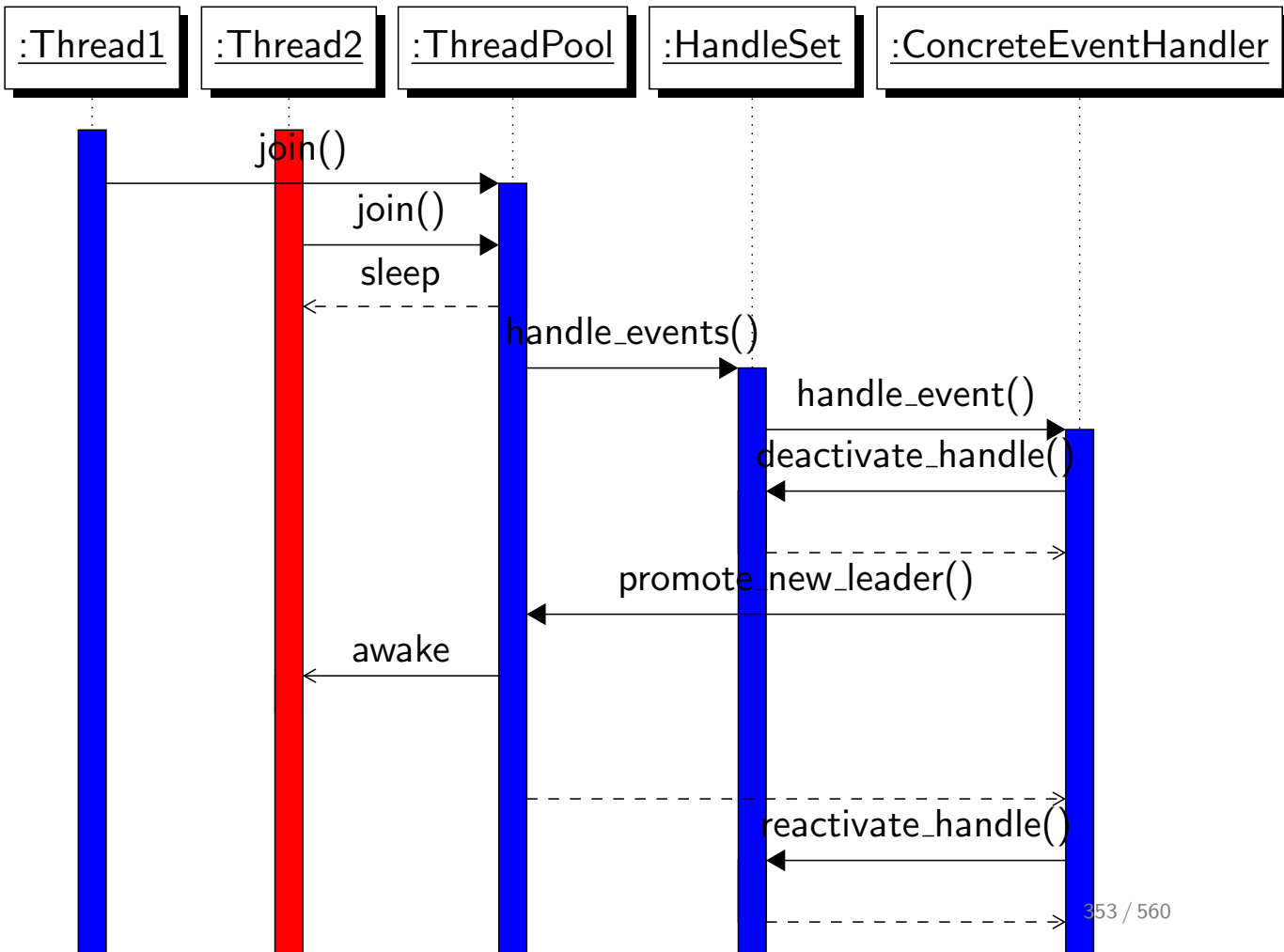



Kontext: Ereignisgesteuerte Anwendung, bei der mehrere Dienstanfragen verschiedener Klienten effizient durch mehrere Threads behandelt werden sollen, die sich die Klienten teilen.
Beispiel: Über ein Netzwerk treffen Anfragen ein, die parallel behandelt werden sollen.
Ein möglicher Entwurf: Ein I/O Thread horcht an einem Socket. Kommt eine Anfrage an, wird sie in eine Queue gesteckt. Die Threads, die die Anfrage bearbeiten, warten, bis Anfragen in der Queue sind, entnehmen diese und bearbeiten sie. Die Queue ist ein Monitor. Der I/O Thread ist ein Produzent und die Dienst-Threads sind Konsumenten.
Nachteil: Es findet ein Kontextwechsel zwischen I/O und Dienstbearbeitung statt.

Entwurfsmuster *Leader/Followers* nach Schmidt u. a. (2000)

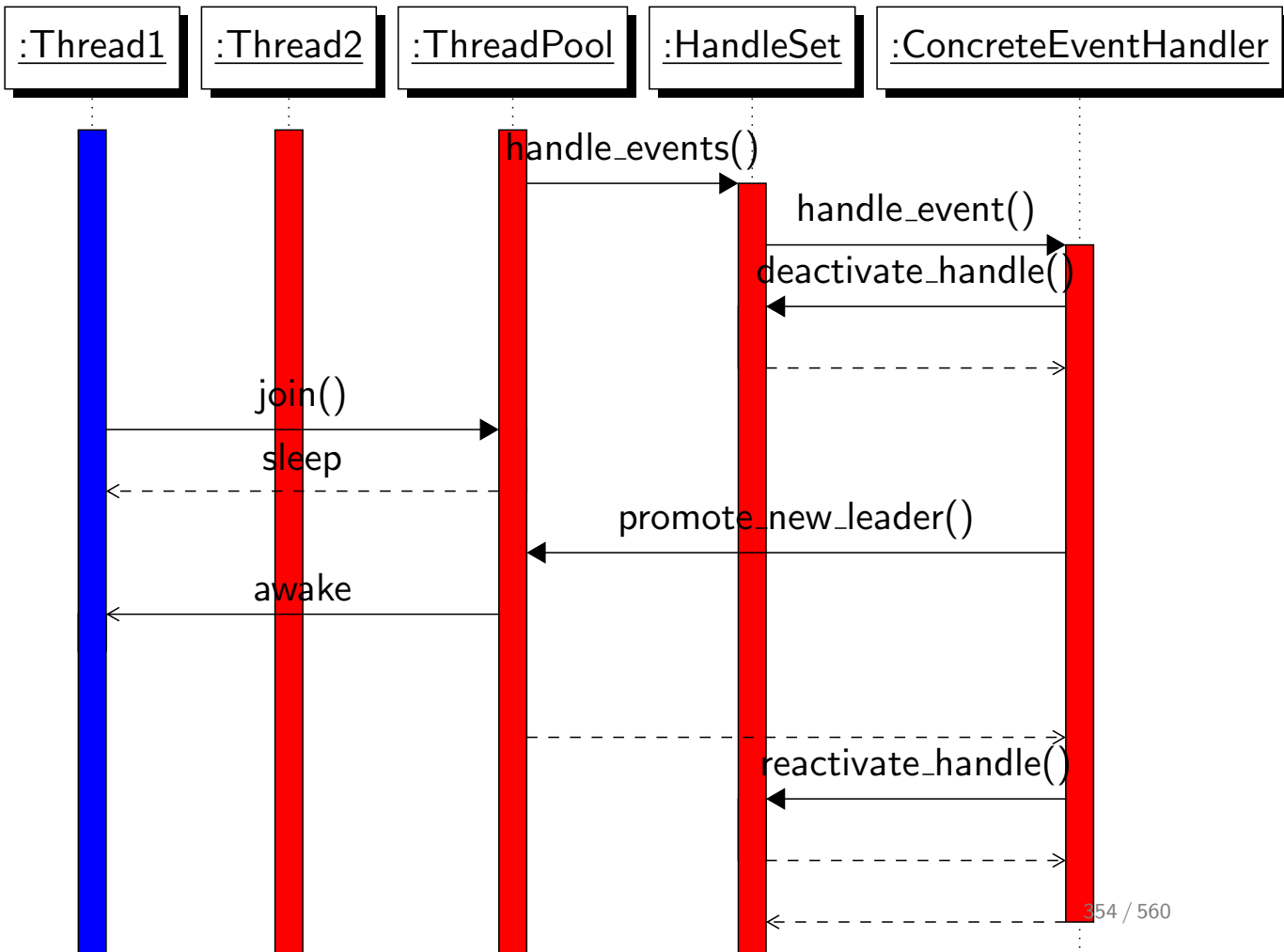


- Handle: identifiziert eine Anfrage (ein Ereignis) über Betriebssystemmechanismen wie Netzwerkverbindungen oder geöffnete Dateien.
- HandleSet: Menge aller Handles.
- EventHandler: Dienst, der für eine Anfrage erbracht werden soll.
- ThreadPool: Menge von Threads, die auf Anfragen warten und sie mit Hilfe eines EventHandler abarbeiten.
- Leader: Thread des ThreadPools, der eine Anfrage erhalten hat und sie abarbeitet.
- Follower: Threads, die bereit sind, als nächstes zum Leader bestimmt zu werden. Bis dahin schlafen sie.



Schritte:

1. Threads melden sich mit `join()` an.
2. Leader wird bestimmt. Andere Prozesse schlafen.
3. Leader wartet auf Ereignis, d.h. auf beliebiges `Handle` im `HandleSet`.
4. Leader nimmt sich dieses `Handle`.
5. Leader bestimmt Nachfolger.
6. Ex-Leader führt Auftrag mittels konkretem `EventHandler` aus.
7. Ex-Leader reicht `Handle` zurück.
8. Ex-Leader tritt mit `join()` dem `ThreadPool` wieder bei.



Server-Code mit mehreren Threads

```
1  // The thread pool where workers will join
2  ThreadPool threadpool(firstport, lastport);
3
4  // Create independent threads
5  std::vector<Poco::Thread*> threads (10);
6  for (int i = 0; i < threads.size(); i++) {
7     threads[i] = new Poco::Thread();
8  }
9
10 std::vector<Worker*> workers;
11 //  each of which will execute function Worker::run().
12 for (int i = 0; i < threads.size(); i++) {
13      Worker* worker = new Worker(&threadpool);
14      workers.push_back(worker);
15      threads[i]->start(*worker);
16 }
17
18 // wait until all threads are finished
19 for (int i = 0; i < threads.size(); i++) {
20      threads[i]->join();
21 }
```


Worker-Thread

```
1  class Worker : public Poco::Runnable {
2  public:
3     Worker(ThreadPool* pool) : threadpool(pool) {};
4     virtual ~Worker() {};
5     void run() {
6        // endless loop, stopped only by killing the process
7        while(true) {
8            threadpool->join(id);
9        }
10    }
11 private:
12     ThreadPool* threadpool;
13 };
```

Thread-Pool

```
class ThreadPool {
public:
    ThreadPool(Poco::UInt16 firstport, Poco::UInt16 lastport);

    void join();
    // to join the thread pool to become a follower or leader;

    void promote_new_leader();
    // a new leader is selected; this method must be called
    // only by the current leader
private:
    Poco::FastMutex mutex;
    // used to block joining threads that are followers

    HandleSet handleset;
};
```


Thread-Pool

```
ThreadPool::ThreadPool(Poco::UInt16 firstport,
                       Poco::UInt16 lastport)
    : handleset(this, firstport, lastport)
{};

void ThreadPool::join()
{
    mutex.lock();
    // if we arrive here, the executing thread
    // becomes the leader
    handleset.handle_events();
}
void ThreadPool::promote_new_leader()
{
    mutex.unlock();
}
```

Handle-Set

```
class HandleSet {
public:
    HandleSet(ThreadPool* pool,
              Poco::UInt16 firstport,
              Poco::UInt16 lastport);
    void handle_events();
private:
    Poco::Net::Socket::SocketList handles;
    ThreadPool* threadpool;
    // the list of handles (sockets) to be listened
    void deactivate_handle(Handle handle);
    void activate_handle(Handle handle);
    Handle select();
};
```



```
HandleSet::HandleSet(ThreadPool* pool,
                     Poco::UInt16 firstport,
                     Poco::UInt16 lastport)
    : threadpool(pool)
{
    // bind to all sockets
    for (int i = firstport; i <= lastport; i++) {
        //creating a socket with Poco
        // a ServerSocket is intended for servers
        Poco::Net::ServerSocket socket(i);
        handles.push_back(socket);
    }
};

void HandleSet::deactivate_handle(Handle handle) {}
// nothing to be done

void HandleSet::activate_handle(Handle handle)   {}
// nothing to be done
```

```
void HandleSet::handle_events() {
  // obtain the handle
  Handle handle = select();

  // this is the handler that handles the event
  EventHandler handler(handle);

  deactivate_handle(handle);

  // we deviate from the leader/follower pattern here
  // and promote the new leader right here and not
  // in the EventHandler
  threadpool->promote_new_leader();

  // handle the request by delegating to the EventHandler
  handler.handle_event();

  activate_handle(handle);
}
```



```
// awaits and handles a request listening on all sockets
// in readSockets
Handle HandleSet::select() {
  // local copy of handles needed because select overwrites
  // its arguments
  Poco::Net::Socket::SocketList readSockets = handles;

  // waits for any request at all readSockets;
  // endless loop because of timeout
  while(true){
    int ready_sockets
        = Poco::Net::Socket::select
            (readSockets, writeSockets, exceptSockets, 100000);
    // readSockets now contains all sockets where requests are
    // available ready_sockets tells how many sockets exist for
    // which we have requests
    if (ready_sockets > 0) break;
  }
  // we handle only one of the requests in readSockets;
  // the others will be served later by the next thread;
  // return the handle (connection) for that request
  return ((Poco::Net::ServerSocket)readSockets[0])
            .acceptConnection();
}
```

Weiterführende Literatur

- Das Standardbuch zu Entwurfsmustern ist das von Gamma u. a. (2003)
- Die Muster für Synchronisation und Parallelisierung stammen von Schmidt u. a. (2000); in dieser Reihe sind weitere Bücher zu Mustern erschienen.

Wiederholungsfragen

- Was ist ein Entwurfsmuster?
- Warum sind sie interessant für die Software-Entwicklung?
- Wie werden Entwurfsmuster von Gamma et al. kategorisiert?
- Erläutern Sie ein in der Vorlesung vorgestellten Entwurfsmuster X (mit Vor- und Nachteilen und Variationen; dabei wird X vom Prüfer vorgegeben).

Komponentenbasierte Entwicklung

Komponentenbasierte Entwicklung
- Lernziele
- Komponente
- Komponentenbasierte Entwicklung
- Komponentenmodelle
- Schnittstellen
- Beschaffung und Entstehung
- Herstellung
- Implementierungsaspekte
- Architektur-Mismatch
- Wiederholungsfragen

Lernziele

- Benutzung von Komponenten
  - Komponentenbasierte Entwicklung
  - Komponentenmodelle
  - Schnittstellen
  - Komponenten
  - Zusammenbau
- Erschaffung von Komponenten
  - Herstellung
  - Implementierungsaspekte

Komponente

Definition

Software components: are executable units of independent production, acquisition, and deployment that interact to form a functioning system (Szyperski u. a. 2002).

- "to compose a system"
- dazu da, um zusammengesetzt zu werden
- N.B.: Komponente $\neq$ Klasse $\neq$ Verteilung

Komponentenbasierte Entwicklung

- Trennung von Schnittstellen (liefern, benutzen) und Implementierung
- Standards für Integration
  - einheitliche Schnittstellenbeschreibung
  - unabhängig von der Programmiersprache
- Infrastruktur (Middleware)
- Entwicklungsprozess

$\rightarrow$ technische und nicht-technische Aspekte

Motivation I

Wiederverwendung als Schlüssel:

- ökonomisch
- als Lösung der Softwarekrise
- OO konnte die Erwartungen an Wiederverwendbarkeit und Vermarktung nicht erfüllen

→ ohne Wiederverwendung: nur lineares Wachstum möglich

→ mit Wiederverwendung: superlineares Wachstum möglich (so die Hoffnung)

Motivation II

Ebenen der Wiederverwendung

- konkrete Lösungsteile: Bibliotheken
- Verträge: Schnittstellen
- Vertragsanbieter: Komponenten
- einzelne Interaktionsteile: Meldungen und Protokolle
- Architekturen für Interaktion: Muster
- Architekturen für Teilsysteme: Frameworks
- Gesamtsystem: Systemarchitekturen

Maßgefertigt vs. Standardsoftware

- **Maßanfertigung:**
  - optimale Anpassung an Kunde → Wettbewerbsvorteil
  - keine Änderung der Kundenprozesse notwendig
  - unter lokaler Kontrolle
- **Standardsoftware:**
  - billiger
  - schneller einsetzbar
  - geringeres Risiko des Scheiterns
  - Wartung und Evolutionsanpassung durch den Hersteller
  - leichtere Zusammenarbeit mit anderen Systemen

Vorteile von beiden Ansätzen: Maßanfertigung aus Standardkomponenten

Integration

Softwerker werden bei komponentenbasierter Entwicklung zu Systemintegratoren:

- Integrationsproblematik wird verschärft:
  - Einbau der Komponenten von Dritten
  - Komponenten kommen aus verschiedenen Quellen
  - Fehlereingrenzung schwierig
  - keine klassischen Integrationstests, da späte Integration
- System nur so stark wie schwächste Komponente

→ defensives Programmieren (bei Hersteller und Verwender) ist zu empfehlen

Vor- und Nachteile I

Vorteile:

- verbesserte Qualität
- Wiederverwendung verringert die Dauer bis zur Auslieferung
- Modularisierung (nur lokale Änderungen, Abhängigkeiten explizit, ...)
- Austauschbarkeit durch Abstraktion (Schnittstellen)
- Markt: innovative Produkte, niedriger Preis,...

Vor- und Nachteile II

Nachteile:

- höhere Kosten durch:
  - komplexere Technik
  - durch Outsourcing höhere Kosten durch Risikoabstützung
  - mehr Aufwand, um vergleichbare Systemstabilität zu erreichen
- offene Probleme:
  - Vertrauenswürdigkeit der Implementierung
  - Zertifizierung der Implementierung
  - gewünschte Anforderungen vs. verfügbare Komponenten

Prozess

Anpassung des Entwicklungsprozesses:

1. Anforderungen erfassen
2. Identifizierung von Komponentenkandidaten (suchen, auswählen, überprüfen)
3. Anpassen der Anforderungen an gefundene Kandidaten
4. Design der Architektur
5. Überprüfung der Komponentenkandidaten und event. neue Suche
6. Erstellen der nichtabgedeckten Funktionalität
7. Zusammenstellen des Systems aus Komponenten mit Verbindungscode (Glue-Code)

Komponentenmodelle

- Sicherstellen der Interoperabilität
- Standards für
  - Schnittstellen:
    - Schnittstellenbeschreibung
    - spezielle Schnittstellen
    - Interaktion zwischen Komponenten
  - Informationen zur Verwendung:
    - Namensregeln
    - Individualisieren
    - Zugriff auf Metadaten
  - Einsatz:
    - Verpackung
    - Dokumentation
    - Evolutionsunterstützung

Beispiele für Komponentenmodelle

- im weiteren Sinn:
  - Anwendungen in einem Betriebssystem
  - Plugins
  - Verbunddokumente (Office Dokumente mit OLE, HTML)
- im engeren Sinn: CORBA, COM, JavaBeans, OSGI

Eigenschaften erfolgreicher Komponentenmodelle

- Infrastruktur mit guter Basisfunktionalität
- Komponenten werden von unterschiedlichen Herstellern angeboten und von Kunden eingesetzt
- Komponenten von verschiedenen Anbietern arbeiten in einer Installation zusammen
- Komponenten haben eine Bedeutung für den Klienten

Allgemeine Dienste

- meist durch Komponentenmodelle als Schnittstelle festgelegt
- Anbieter der Komponentenmodelle bietet auch diese Komponenten an
- besonders wichtig für verteilte Systeme
- Beispiele:
  - Verzeichnisdienst
  - Persistenz
  - Nachrichtendienst
  - Transaktionsmanagement
  - Sicherheit

Schnittstellen

- Schnittstellen ermöglichen:
  - Zusammenarbeit zwischen fremden Komponenten
  - Austauschbarkeit (Anbieter und Benutzer)
  - Identifizierung der Abhängigkeiten
- Interna bleiben verborgen (alle Zugriffe über Schnittstelle)
- Qualität von höchster Bedeutung
- eine Komponente kann Serviceprovider für mehr als eine Schnittstelle sein (provides)
- eine Komponente kann den Service anderer Schnittstellen benötigen (requires)
- späte Integration $\rightarrow$ späte Bindung $\rightarrow$ Indirektion
- beschrieben durch Meta-Information zur Laufzeit, Interface Description Language (IDL) oder „direkt“ in Programmiersprache

Beispiel (CORBA-IDL)

```
module Bank {
  typedef long pin_t;
  enum KontoFehlerTyp {
    UngenuegendeKontodeckung
  };

  exception KontoException {
    KontoFehlerTyp typ;
    string beschreibung;
  };

  interface Konto {
    readonly attribute string name;
    readonly attribute long kontoStand;

    boolean isValidPin(in pin_t pin);
    void abheben(in long betrag) raises(KontoException);
    void einzahlen(in long betrag);
  };
};
```


Vertrag

- Schnittstelle als Vertrag zwischen Anbieter und Benutzer des Services
- Anbieter:
  - über Funktionalität: z.B. als Vor- und Nachbedingungen
  - über nicht-funktionale Anforderungen (Service-Level, Ressourcen);
    z.B. Standard Template Library für C++
  - Darstellung:
    - informal als Text
    - formaler z.B. durch temporale Logik (um Terminierung zuzusichern) oder mit OCL
- Kunde:
  - Vermeidung von speziellen Eigenschaften einer bestimmten Implementierung (d.h. nur Vertrag benutzen)

Versionen

- Problem: sowohl Schnittstelle als auch Implementierung ändern sich $\rightarrow$ unterschiedliche Versionen nicht vermeidbar
- Ziel: Entscheidung, ob kombatibel oder nicht
  - zusätzlich noch Unterstützung für einen Bereich von Versionen (sliding window)
- Lösungen (Lösungen?):
  - unveränderliche Schnittstellen
  - Schnittstellen dürfen sich ändern, aber nur nach Regeln (z.B. Parametertyp darf verallgemeinert werden)
  - Ignorieren des Problems:
    - abwälzen auf tiefere Schicht
    - immer neu kompilieren

Beschaffung

- Komponenten werden nach funktionalen und nichtfunktionalen Anforderungen klassifiziert
- Qualitätskriterien für Auswahl:
  - Erfüllungsgrad funktionaler und nichtfunktionaler Anforderungen
  - Schnittstellenbeschreibung
  - Abhängigkeiten und Kompatibilität
  - Vertrauenswürdigkeit, Überlebenschance des Anbieters, Garantien und Wartungszusicherung
  - Preis und Zahlungsart
  - ...

Entstehung von Komponenten

- meist aus bestehender Software
- Anpassung notwendig, um Wiederverwendbarkeit zu erreichen
- ist Investition ökonomisch sinnvoll?
  - Größe des Einsatzgebiets
  - bislang angebotene Funktionalität
  - potentieller Grad der Wiederverwendung

Balance zwischen

großen Komponenten

- bieten mehr Service an
- weniger Abhängigkeiten
- schneller, da keine cross-context Aufrufe

kleinen Komponenten

- verständlicher
- billiger für den Benutzer
- mehr Freiheit für den Benutzer
- mehr Benutzer

Implementierung

- defensives Programmieren, da keine Integrationstests
- existierenden Code wiederverwendbar machen:
  - entferne anwendungsspezifische Methoden
  - generalisiere Namen
  - füge Methoden hinzu, um Funktionalität zu vervollständigen
  - führe konsistente Ausnahmebehandlung ein
  - füge Möglichkeiten hinzu, die Komponenten an verschiedene Benutzer anzupassen
  - binde benötigte Komponenten ein, um Unabhängigkeit zu erhöhen

Typen

- Typ als vereinfachter Vertrag
- syntaktische Überprüfung durch Compiler
- semantische Überprüfung durch explizite Vor- und Nachbedingungen (Übersetzungszeittest besser als Ladezeittest besser als Laufzeittest besser als kein Test)
- Implementierungen dürfen Vorbedingungen abschwächen oder Nachbedingungen verschärfen

Vererbung

- Anpassen einer Komponente:
  - durch Parametrisieren und Verbinden mit anderen Komponenten
  - durch Ableiten von Subtypen
- Arten:
  - Implementierungsvererbung
  - Schnittstellenvererbung
- Subtypen
  - Austauschbarkeit (Liskovsches Substitutionsprinzip); deklarative vs. strukturelle Subtypen
  - Vererbung von Parametertypen in Signatur `foo(T)` in Klasse T:
    - Invarianz: selber Typ T
    - Kontravarianz: Supertyp von T
    - Kovarianz: Subtyp von T
- Mehrfachvererbung:
  - Schnittstellenvererbung: kein Problem
  - Implementierungsvererbung: problematisch

Zerbrechliche Basisklassen

- Basisklasse wird geändert, sind erbende Klassen immer noch funktionsfähig?
- unvorhergesehene Aufrufgraphen

```
// Version 1
class T {
    public void foo() {}
}

// Client code
class NT {
  public void bar() {}
}
```

```
// Version 2
class T {
    public void foo() {bar();}
    private int i;
    public void bar() { i = 1;}
}
```


Zerbrechliche Basisklassen

- notwendig ist Schnittstellenvertrag zwischen Klasse und erbenden Klassen
- Lösungen:
  - Ableiten der Klasse verbieten (z.B. `final`)
  - Sichtbarkeitsschutz (`public`, `protected`, `private`, `final`, `override`)
  - Offenlegung der Funktionsweise der Basisklassen und Regeln, was Unterklassen dürfen

Aufrufe zwischen Komponenten in verteilten Systemen

- Idee des lokalen Aufrufes bleibt erhalten
- Generierung von Stubs
- Aktionen des Aufrufers bei Aufruf:
  1. Kontrolle geht an Stub
  2. Marshalling/Serialisierung
  3. Übertragung der Parameter
  4. Ausführen auf dem Ziel
  5. Übertragung des Ergebnisses/Ausnahme
  6. Unmarshalling
  7. Rückgabe an den Aufrufer
- Optimierung für lokalen Fall

Wiedereintritt

```
int global = 1;

int foo()
{// non reentrant
 global = global + 1; // what if interrupted here?
 return global;
}

int bar()
{// non reentrant (transitivity)
 return foo() + 1;
}
```

- Vorkommen: Callbacks (von der unteren zur oberen Schicht) oder Multithreading
- Problem: welche Funktionen dürfen wann benutzt werden? Beispiel: Dokumentation zu Unix-Signal-Handler
- nicht mit Vor- und Nachbedingungen ausdrückbar

Fallstudie zur komponentenbasierten Entwicklung

Garlan u. a. (1995): Komposition eines Case-Tools aus

- einer objektorientierten Datenbank
- Toolkit zur Konstruktion graphischer Benutzeroberflächen
- event-basiertem Tool-Integrations-Mechanismus
- RPC-Mechanismus

Alle Komponenten in C oder C++ geschrieben.

Glaube und Wirklichkeit

Schätzung:

- Dauer: 6 Monate
- Aufwand: 12 PM

Tatsächlich:

- Dauer: 2 Jahre
- Aufwand: 60 PM für ersten Prototyp

Ergebnis:

- sehr großes System
- träge Performanz
- viele Anpassungen für die Integration notwendig
- existierende Funktionalität musste neu implementiert werden, weil sie nicht exakt den Anforderungen entsprach

Architektur-Mismatch

Definition

Architektur-Mismatch: inkompatible Annahmen von wiederzuverwendenden Komponenten über das Systems, in dem sie eingesetzt werden sollen.

Meist spät entdeckt, weil die Annahmen in aller Regel nur implizit sind.

Komponenten betreffend

Annahmen von Komponenten über

- ihre Infrastruktur, die zur Verfügung gestellt werden soll bzw. vorausgesetzt wird
  - Komponenten stellten vieles zur Verfügung, was gar nicht gebraucht wurde

  → exzessiver Code
- das Kontrollmodell: wer steuert den Kontrollfluss
  - jede Komponente nahm an, dass sie die Hauptkontrollschleife darstelle

  → aufwändige Restrukturierung notwendig
- das Datenmodell: Art und Weise, wie die Umgebung Daten manipuliert, die von der Komponente verwaltet werden
  - hierarchische Datenstruktur erlaubte Änderung der Teile nur über das Ganze

  → für Anwendung zu unflexibel

  → teilweise Neuimplementierung

Konnektoren betreffend

Annahmen von Konnektoren über

- Protokoll: Interaktionsmuster
  - Semantik des synchronen Aufrufs passte nicht

  → Ausweichung auf RPC des Betriebssystems hierfür
- Datenmodell: Art der Daten, die kommuniziert werden
  - RPC des Betriebssystems nahm an, C-Datenstrukturen zu transportieren
  - wiederzuverwendendes Event-Broadcast-System nahm an, ASCII zu transportieren

  → Konvertierungsroutinen wurden notwendig

Architekturkonfiguration betreffend

Annahmen über Architekturkonfiguration

- Topologie der Systemkommunikation
  - Datenbank nahm an, dass verbundene Tools nicht kooperieren wollen und blockierte sie, um Sequenzialisierung zu garantieren
  - Tools mussten aber kooperieren

  → eigener Transaktionsmonitor musste implementiert werden
- An- oder Abwesenheit bestimmter Komponenten und Konnektoren

Konstruktionsprozess betreffend

Annahmen über Konstruktionsprozess (wie Komponenten/Konnektoren aus generischen Einheiten erstellt werden – sowohl zur Übersetzungs- als auch zur Laufzeit)

Beispiele:

- Datenbank → Schema muss festgelegt werden
- Event-System → Menge der Ereignisse und Registration

Annahmen über die Reihenfolge, in der Teile erstellt und kombiniert werden.

- nicht zueinander passende Annahmen über den Konstruktionsprozess

→ Umwege machten Konstruktionsprozess aufwändig und kompliziert

Wiederholungs- und Vertiefungsfragen

- Was ist eine Komponente?
- Was ist die Idee der komponentenbasierten Entwicklung? Welche Ziele verfolgt sie?
- Diskutieren Sie Vor- und Nachteile maßgefertigter Software versus Software von der Stange.
- Was ist ein Komponentenmodell? Was wird von einem solchen üblicherweise festgelegt bzw. zur Verfügung gestellt?
- Was ist eine Schnittstelle und welche Bedeutung kommt diesem Konzept im Kontext komponentenbasierter Entwicklung zu?
- Wie können vorgefertigte Komponenten angepasst werden?
- Welches Problem tritt bei Anpassung durch Vererbung und Redefinition auf? Wie kann man mit ihm umgehen?
- Was ist das Problem des Wiedereintritts?
- Was versteht man unter Architektur-Mismatch und welche Bedeutung hat er für die komponentenbasierte Entwicklung?

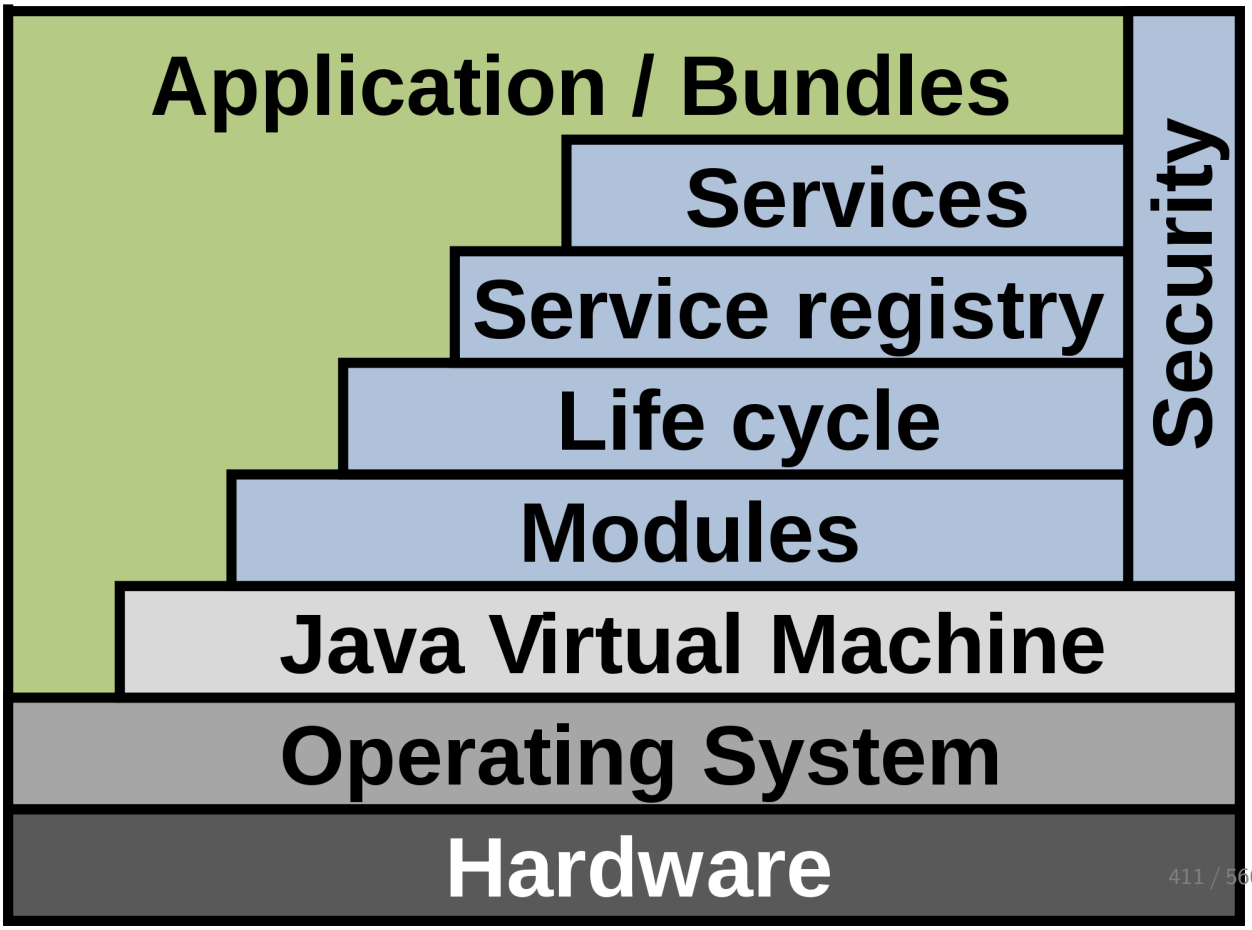
OSGI als Beispiel eines Komponentenmodells
Architektur
Manifest
Lebenszyklus von Bundles
Existierende Container-Implementierungen
Demo mit Eclipse
Wiederholungsfragen

Beispiel eines Komponenten-Rahmewerks: OSGI

Open Services Gateway Initiative (OSGI)

- Kompontenmodell für Java
- Komponente = *Bundle* = JAR + Manifest
- unterstützt entfernte Installation, Start, Stopp, Aktualisierung und Deinstallation von *Bundles*
- Service-Register ermöglicht den Bundles, Dienste hinzuzufügen, zu entfernen und anzupassen
- Verbreitung: Eclipse, Mobiltelefone, ...

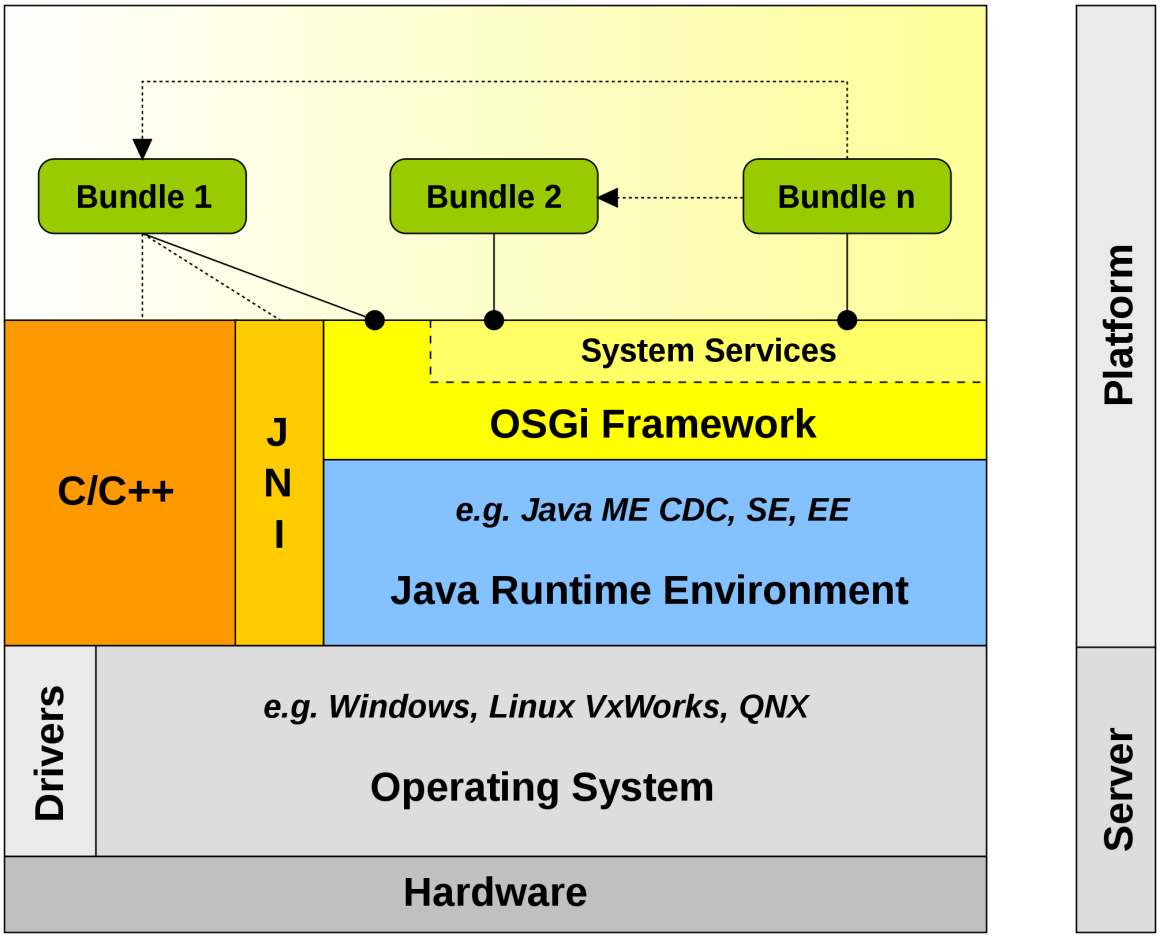
Architektur



Bundles	Bundles are normal jar components with extra manifest headers.
Services	The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java Interfaces (POJI) or Plain Old Java Objects POJO.
Services Registry	The API for management services (ServiceRegistration, ServiceTracker and ServiceReference).
Life-Cycle	The API for life cycle management for (install, start, stop, update, and uninstall) bundles.
Modules	The layer that defines encapsulation and declaration of dependencies (how a bundle can import and export code).
Security	The layer that handles the security aspects by limiting bundle functionality to pre-defined capabilities.

Bildquelle: Faisal Akeel, Bill Streckfus

Architektur



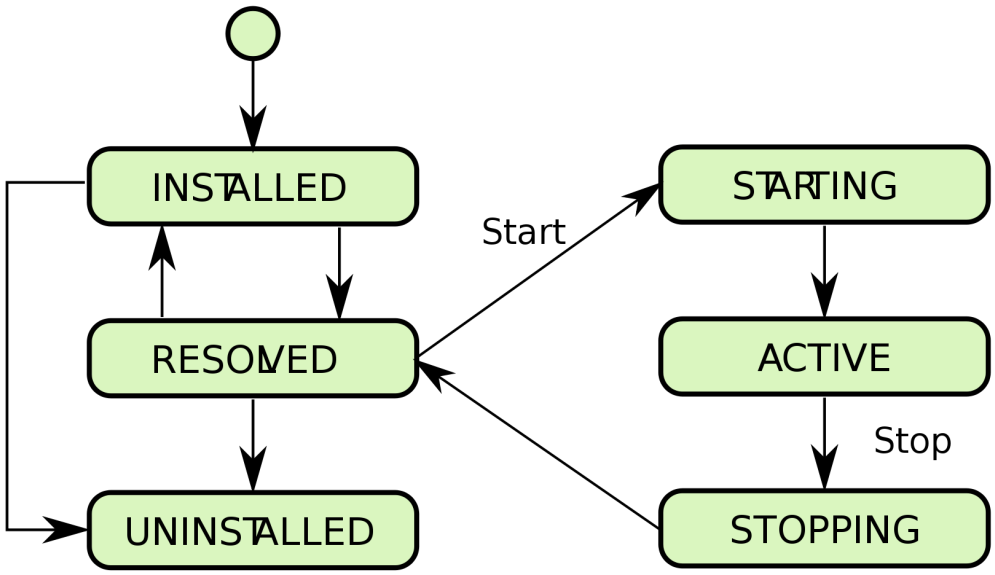
Bildquelle: Michael Grammling, Bill Streckfus (Creative Commons Attribution ShareAlike 3.0 License)

Bundle-Manifest

```
Bundle-Name: Hello World
Bundle-SymbolicName: org.wikipedia.helloworld
Bundle-Description: A Hello World bundle
Bundle-ManifestVersion: 2
Bundle-Version: 1.0.0
Bundle-Activator: org.wikipedia.Activator
Export-Package: org.wikipedia.helloworld;version="1.0.0"
Import-Package: org.osgi.framework;version="1.3.0"
```


Bundle-Name	Defines a human-readable name for this bundle, Simply assigns a short name to the bundle.
Bundle-SymbolicName	The only required header, this entry specifies a unique identifier for a bundle, based on the reverse domain name convention (used also by the java packages).
Bundle-Description	A description of the bundle's functionality.
Bundle-ManifestVersion	This little known header indicates the OSGi specification to use for reading this bundle.
Bundle-Version	Designates a version number to the bundle.
Bundle-Activator	Indicates the class name to be invoked once a bundle is activated.
Export-Package	Expresses what Java packages contained in a bundle will be made available to the outside world.
Import-Package	Indicates what Java packages will be required from the outside world, in order to fulfill the dependencies needed in a bundle.

Lebenszyklus eines Bundles



INSTALLED	The bundle has been successfully installed.
RESOLVED	All Java classes that the bundle needs are available. This state indicates that the bundle is either ready to be started or has stopped.
STARTING	The bundle is being started, the BundleActivator.start method will be called, and this method has not yet returned. When the bundle has an activation policy, the bundle will remain in the STARTING state until the bundle is activated according to its activation policy.
ACTIVE	The bundle has been successfully activated and is running; its Bundle Activator start method has been called and returned.
STOPPING	The bundle is being stopped. The BundleActivator.stop method has been called but the stop method has not yet returned.
UNINSTALLED	The bundle has been uninstalled. It cannot move into another state.

Bildquelle: Faisal Akeel

Open-Source-OSGi-Container

- Equinox
- Knopflerfish
- Apache Felix

Equinox is the reference implementation for the framework portion of the OSGi Service Platform Release 4. It is the modular Java runtime at the heart of the Eclipse IDE, and implements all of the mandatory and most of the optional features of the OSGi R4 specification.
Knopflerfish is an open source implementation of the OSGi R3 and OSGi R4 specifications. Knopflerfish 2 implements all the mandatory features and some of the optional features defined in the R4 specification.
Apache Felix is the open source OSGi container from the Apache Software Foundation. At the time of writing this container is not fully compliant with the OSGI R4 specification.

Quelle: Wikipedia

Eclipse-Demo: Einfaches Bundle anlegen

1. Neues Plug-in-Projekt "File → New → Project"
2. Selektiere "Plug-in Project" und "Next".
3. Eingabe:
   - Project Name: com.javaworld.sample.HelloWorld
   - Target Platform: OSGi framework → Equinox
4. Weiter mit "Next".
5. Selektiere Voreinstellungen im Plug-in-Context-Dialog und "Next"
6. Templates-Dialog "Hello OSGi Bundle" und "Finish"
7. Manifest anschauen.
8. Quellcode von Activator anschauen

Eclipse-Demo: Einfaches Bundle starten

1. Framework testweise starten: Sektion "Testing", Link "Launch the framework"
2. Status des Bundles anschauen: "ss Hello" in OSGI-Konsole eingeben.
3. Bundle anhalten: "stop <nummer>"
4. Bundle starten: "start <nummer>"
5. Quelltext von Activator verändern: Ausgabe verändern.
6. Bundle aktivieren: "update <nummer>"

Eclipse-Demo: Export

1. Neues Plug-in-Projekt com.javaworld.sample.HelloService anlegen (wie oben)
2. Im Projekt neues Interface *HelloService* im Package *com.javaworld.sample.service*:

```
package com.javaworld.sample.service;
public interface HelloService {
        public String sayHello();
}
```

3. Neue Klasse *HelloServiceImpl* im Package *com.javaworld.sample.service.impl*:

```
package com.javaworld.sample.service.impl;
import com.javaworld.sample.service.HelloService;

public class HelloServiceImpl implements HelloService {
        @Override
        public String sayHello() {
                return "at your service";
        }
}
```


4. In MANIFEST.MF im Reiter *Runtime* als *Exported Packages* *com.javaworld.sample.service* hinzufügen

Eclipse-Demo: Import

1. In MANIFEST.MF von Plug-In HelloWorld im Reiter *Required Plug-ins* *com.javaworld.sample.HelloService* hinzufügen
2. Editiere com.javaworld.sample.helloworld.Activator.java: HelloService-Interface ist sichtbar, aber nicht HelloServiceImpl-Klasse:

```
public void callService(HelloService service) {
   System.out.println("HelloWorld: " + service.sayHello());
}
```

Eclipse-Demo: Service-Provider

Im Plug-in-Projekt *com.javaworld.sample.HelloService* die Klasse *Activator wie folgt abändern*:

```
package com.javaworld.sample.helloservice;
import org.osgi.framework.BundleActivator;

public class Activator implements BundleActivator {

    ServiceRegistration helloServiceRegistration;
    public void start(BundleContext context) throws Exception {
        System.out.println("Service: Hello World!!");
        HelloService helloService = new HelloServiceImpl();
        helloServiceRegistration
          = context.registerService(HelloService.class.getName(),
                                    helloService, null);
    }
    public void stop(BundleContext context) throws Exception {
        System.out.println("Service: Goodbye World!!");
        helloServiceRegistration.unregister();
    }
}
```


In OSGi, a source bundle registers a POJO (you don't have to implement any interfaces or extend from any superclass) with the OSGi container as a service under one or more interfaces. The target bundle can then ask the OSGi container for services registered under a particular interface. Once the service is found, the target bundle binds with it and can start calling its methods.

Quelle: `http://www.javaworld.com/javaworld/jw-03-2008/jw-03-osgi1.html?page=4`

Eclipse-Demo: Service-Consumer

Im Plug-in-Projekt *com.javaworld.sample.HelloWorld* die Klasse *Activator wie folgt abändern*:

```
public class Activator implements BundleActivator {

  ServiceReference helloServiceReference;
  public void start(BundleContext context) throws Exception {
    System.out.println("Consumer: Hello World!!");
    helloServiceReference
     = context.getServiceReference(HelloService.class.getName());
    HelloService helloService
     = (HelloService)context.getService(helloServiceReference);
    System.out.println(helloService.sayHello());
  }
```


Eclipse-Demo: Service-Consumer (Forts.)

```
  public void stop(BundleContext context) throws Exception {
     System.out.println("Consumer: Goodbye World!");
     context.ungetService(helloServiceReference);
  }
}
```

Eclipse-Demo: Start von Service-Provider/Consumer

1. Im Manifest von *com.javaworld.sample.HelloService* “Launch the framework” auswählen.
2. “ss Hello”
3. “update <nummer>”, wobei *nummer* die Id des Consumer-Plug-in *HelloWorld* ist

Wiederholungs- und Vertiefungsfragen

- Erläutern Sie die Aspekte eines Komponentenmodells anhand von OSGI?
- Welche Art der Schnittstelleninformation ist in einer Manifest-Datei von OSGI enthalten? Welche Informationen der Schnittstelle sind hingegen nur im Programmcode erkennbar?
- Beschreiben Sie den Lebenszyklus eines Bundles in OSGI.
- Wie können Services eines Bundles von anderen Bundles aufgerufen werden?

Modellgetriebene Softwareentwicklung
- Modellgetriebene Entwicklung
- Modelle
- Metamodelle
- Syntax und Semantik von Modellen
- Standards
- Domänenspezifische Sprachen (DSL)
- Werkzeuge für DSLs
- Modelltransformationen
- Modell-zu-Text-Transformationen
- Zusammenfassung
- Wiederholungsfragen

Definition

Modellgetriebene Softwareentwicklung (Model-Driven Software Development (MDSD)) bezeichnet Softwareentwicklungsprozesse, bei denen Modelle im Mittelpunkt stehen und als eigenständige Entwicklungsartefakte genutzt werden (Reussner und Hasselbring 2009).

Modelle sind zentrale Entwicklungsartefakte:

- Kommunikation mit Fachexperten
- Analyse
- Generierung von Code

Ziele: Reduktion der Komplexität durch

- Abstraktion (Reduktion auf das Wesentliche)
- Automatisierung

- Handhabung von Komplexität
  - Abstraktion zum Wesentlichen
  - Einbindung von Fachexperten
  - Trennung von Aufgaben und Belangen
- Steigerung der Entwicklungseffizienz
  - Generierung von redundantem Programmcode
  - Wiederverwendung
- Steigerung der Softwarequalität
  - Wohldefinierte Regeln für Modelle
  - Bewährte Transformationen
  - Homogener Programmcode
- Interoperabilität und Portabilität

Merkmale eines Modells nach Stachowiak (1973)

Abbildung

- Repräsentation eines betrachteten Gegenstands
- Übertragbarkeit bestimmter Beobachtungen am Modell auf modellierten Gegenstand

Verkürzung

- Betrachtung der *relevanten* Attribute des betrachteten Systems im Modell für bestimmte Betrachter
- irrelevante Attribute werden nicht repräsentiert

Pragmatismus

- das Modell ist einem bestimmten Zweck zugeordnet
- der Zweck bestimmt Verkürzung und Abbildung

Modell und Metamodell

Definition

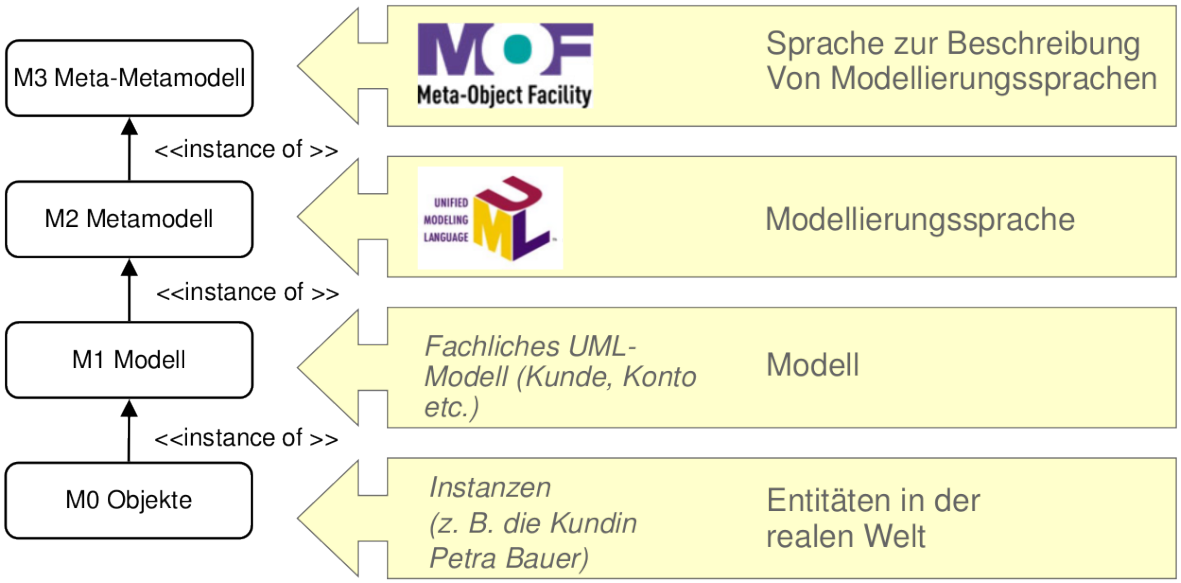
Ein **Metamodell** ist das Modell einer Menge von Modellen.

– Favre (2004)

Definition

Ein **Metametamodell** ist das Modell einer Menge von Metamodellen.

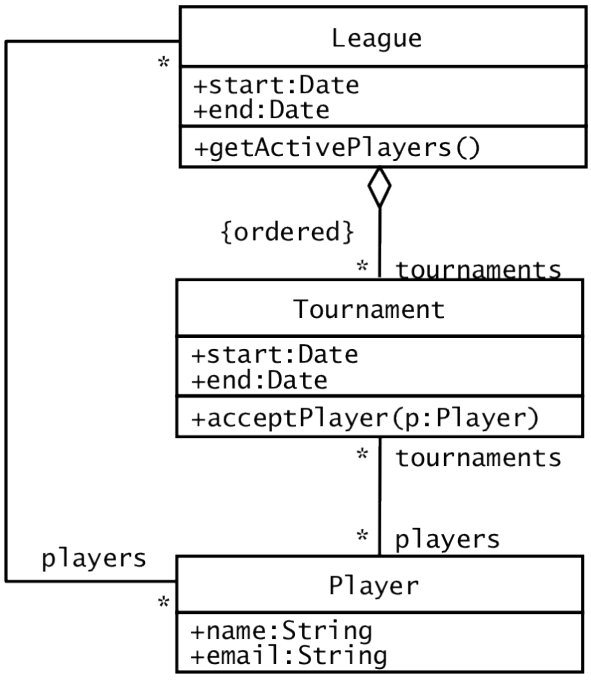
Ein Modell ist selbst ein Betrachtungsgegenstand und kann modelliert werden.Ein Metamodell ist selbst wieder ein Modell und wird durch ein Metamodell beschrieben: ein Metametamodell.



Syntax und Semantik von Modellen (Stahl u. a. 2007)

Konkrete Syntax

- Definiert die konkrete Darstellung von Modellen
- Regeln für die Abbildung auf die abstrakte Syntax



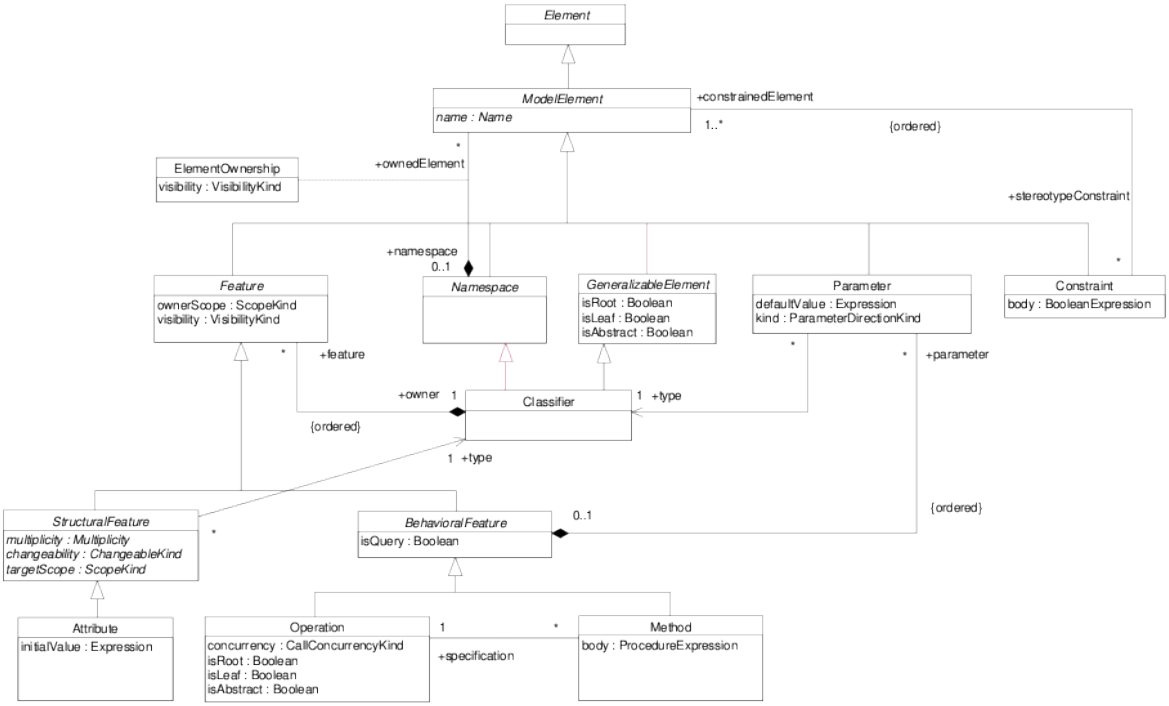
Konkrete Syntax: Eine Klasse wird als Rechteck gezeichnet. . . Bildquelle: OCL Tutorial von Mazeiar Salehie

http://www.stargroup.uwaterloo.ca/~ltahvild/courses/ECE493-T5/tutorials/Tutorial-Feb16-OCL.pdf

Abstrakte Syntax

- Definiert den Aufbau korrekter Instanzen
- Elemente und ihre Beziehungen

Abstrakte Syntax von UML-Klassendiagrammen (Ausschnitt)



Class leitet von *Classifier* ab.

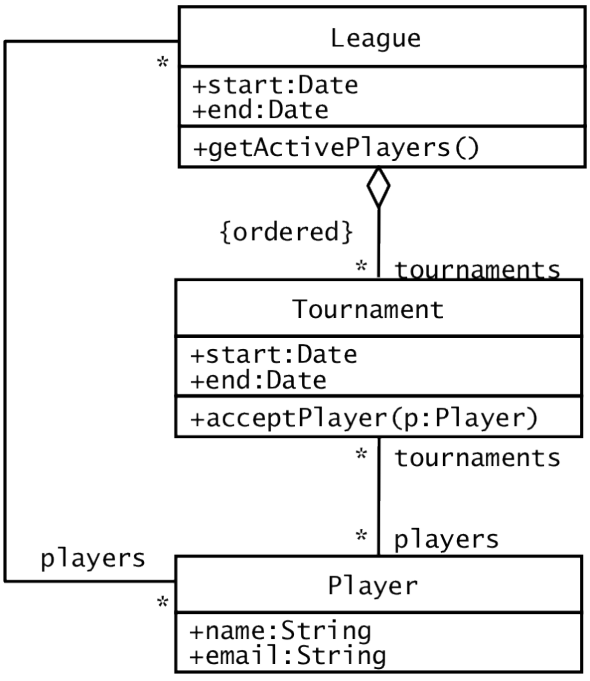
Syntax und Semantik nach Stahl u. a. (2007)

Statische Semantik

- Schränkt die abstrakte Syntax ein

OCL-Beispiel:

context Tournament
inv: end - start <=
Calendar.WEEK



Statische Semantik: Der Name einer Klasse muss eindeutig sein. Kann z.B. mit Object Constraint Language (OCL) ausgedrückt werden. Bildquelle und OCL-Beispiel: OCL Tutorial von Mazeiar Salehie

`http://www.stargroup.uwaterloo.ca/~ltahvild/courses/ECE493-T5/tutorials/Tutorial-Feb16-OCL.pdf`

Syntax und Semantik nach Stahl u. a. (2007)

„Dynamische“ Semantik

- Bedeutung bzw. Interpretation der abstrakten Syntax
- z.B. formalisiert als Abbildung der abstrakten Syntax auf ein mathematisches Modell (denotionale Semantik)
- z.B. formalisiert als Abbildung auf ausführbares Modell (z.B. Code) (operationale Semantik)

„Dynamische“ Semantik: Jede Instanz der Klasse hat alle Attribute ihrer Klasse. Methoden der Klasse können diese lesen und schreiben.

Standards der modellgetriebenen Entwicklung

- Model Driven Architecture (MDA)
- UML
- MOF

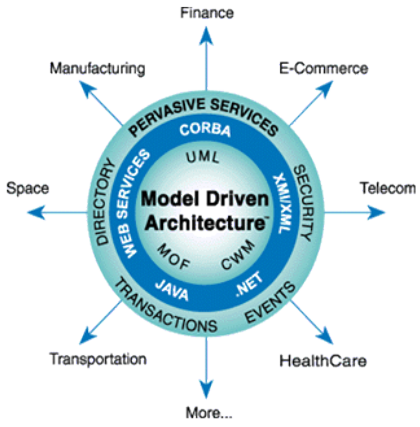
Standard: Model Driven Architecture (MDA)

MDA: Ansatz der Object Management Group (OMG) zur modellgetriebenen Entwicklung mit den Zielen:

- Interoperabilität
- Portabilität
- Wiederverwendbarkeit

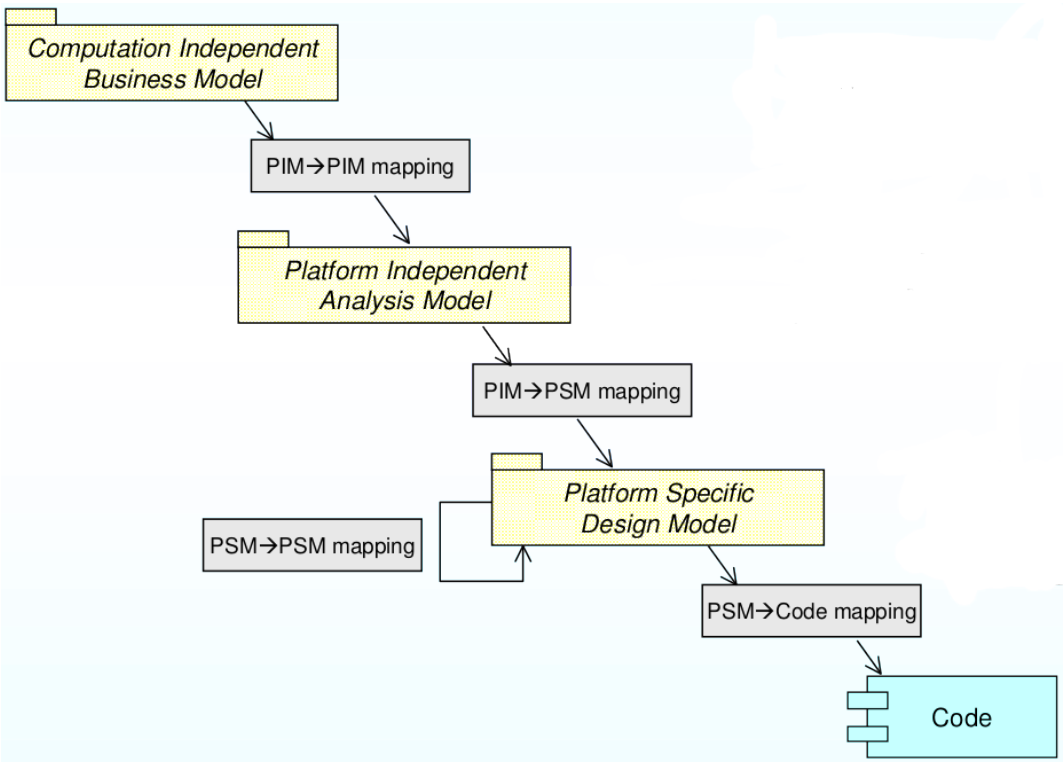
Verbindet verschiedene OMG-Standards

- Meta Object Facility (MOF)
- Unified Modeling Language (UML)
- Common Warehouse Model (CWM)
- Query/Views/Transformations (QVT)
- XML Metadata Interchange (XMI)



Bildquelle: `http://www.omg.org`

Model-Driven Architecture



Computation Independent Model (CIM):

- Modell der Domäne
- Enthält die Anforderungen an das zu entwickelnde System

Platform Independent Model (PIM)

- Modell der Implementierung unabhängig von der Zielplattform
- Grundlage für die Entwicklung eines Systems auf verschiedenen Zielplattformen

Platform Specific Model (PSM)

- Ergänzt das PIM mit Details zu einer bestimmten Plattform
- Evtl. zusätzlich Platform Model zwischen PIM und PSM

Code

- Ausführbarer Quellcode

Standard: UML 2.x

UML-Infrastructure

- Grundlagen der abstrakten Syntax
- z.B. Klassen, Assoziationen, Multiplizitäten

UML-Superstructure

- Erweiterungen der UML-Infrastructure um Elemente für spezielle Modellierungsaufgaben
- z.B. Komponenten, Anwendungsfälle, Aktivitäten

Object Constraint Language (OCL)

- Beschreibung der statischen Semantik
- Invarianten, Vor- und Nachbedingungen etc.

Diagram Interchange (XMI)

- Austausch der grafischen Modellrepräsentation
- z.T. uneinheitlich implementiert

Standard: Meta Object Facility (MOF)

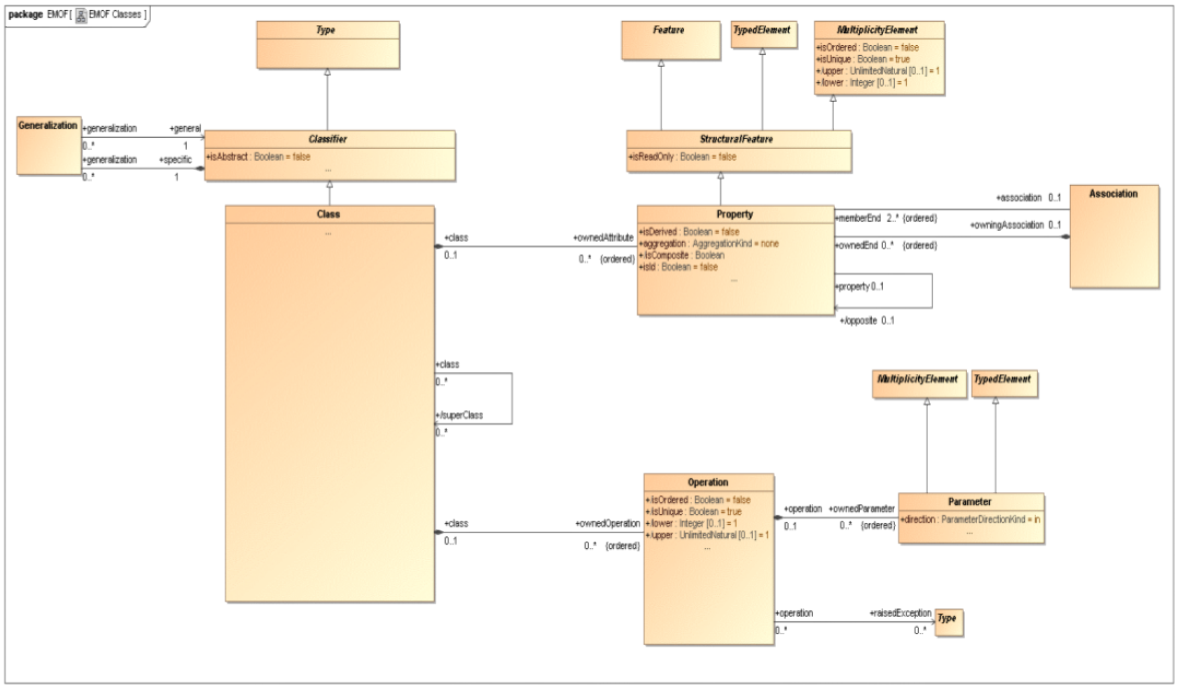
MOF: Standard der Object Management Group zur Metamodellierung
$\rightarrow$ basiert auf UML-Klassendiagrammen

Varianten:

- Complete MOF (CMOF):
  Gesamter Sprachumfang von MOF
- Essential MOF (EMOF):
  - Minimale Untermenge von MOF, um Metamodelle beschreiben zu können
  - Weitestgehend kompatibel zu *Ecore* aus dem Eclipse Modeling Framework (EMF)

Bildquelle: `http://www.omg.org/mof/`

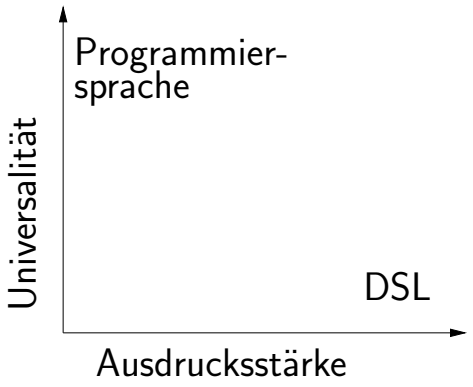
EMOF



Domänenspezifische Sprache

Definition

Domänenspezifische Sprache (Domain-specific Language DSL): Sprachen, die auf eine spezielle Anwendungsdomäne zugeschnitten sind.



They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application. *(Mernik u. a. 2005)*

DSLs tauschen Universalität (allgemeine Anwendbarkeit) gegen Ausdrucksstärke in einer begrenzten Domäne

Arten von DSLs

Interne DSLs (Language Exploitation): eingebettet in bestehende Sprachen

- Nutzung einer existierenden Wirtssprache (Piggyback)
- Spezialisierung und Erweiterung der Wirtssprache, z.B. UML-Profile als Spezialisierung
- Nutzung bestehender Werkzeuge

Externe DSLs (Language Invention)

- Grammatik (abstrakte Syntax, Metamodell)
- Notation (konkrete Syntax, grafisch oder textuell)
- Benötigen eigene Werkzeuge

Repräsentation von DSLs

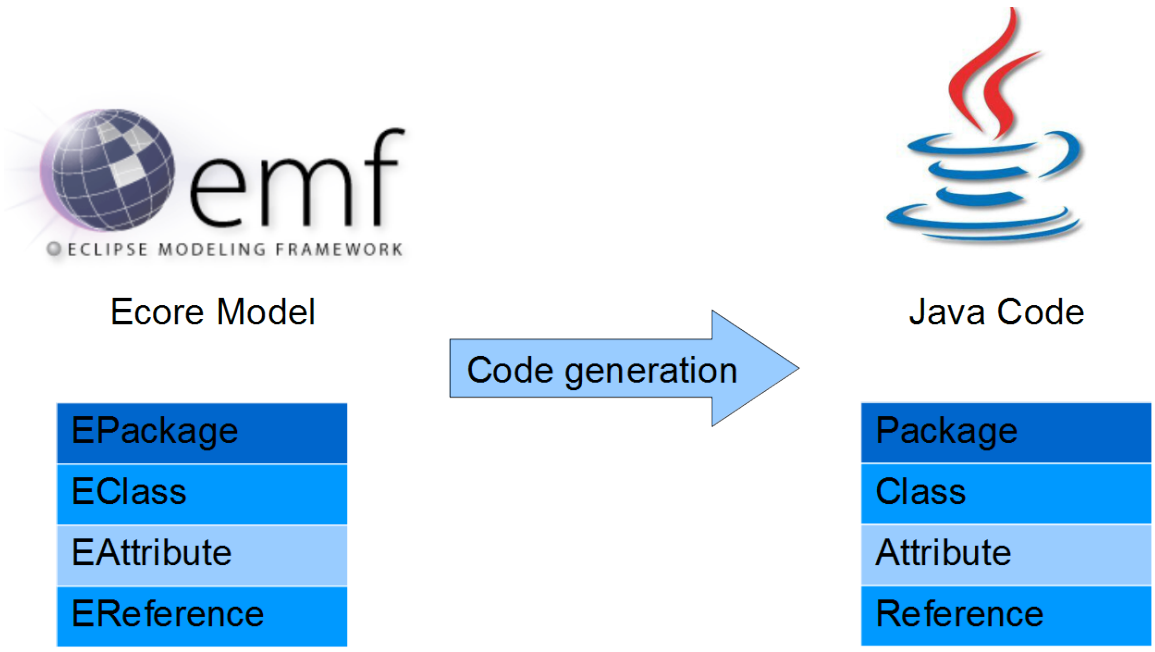
Grafische Notation (Rendering)

- Hohe Informationsdichte
- Mehrdimensionalität (Kleppe 2008)
- Häufig graphorientiert (Knoten & Kanten)
- Bei größeren Projekten abwägen:
  Aufwand Layout > Aufwand Modellierung?

Textuelle Notation (Serialisierung)

- Schnell editierbar, breite Werkzeugunterstützung
- Häufig sehr kompakt und formal
- Häufig blockstrukturiert (Textabschnitte bilden Blöcke)
- Darstellung von Beziehungen zwischen Entitäten schwierig (z.B. Verweise)

Eclipse Modeling Framework (EMF)

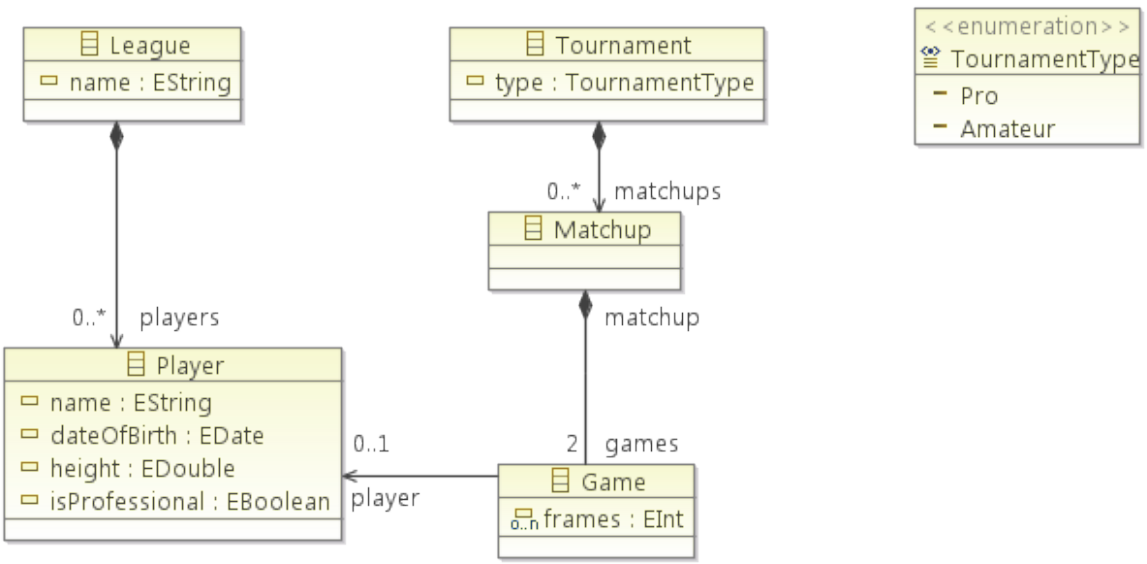


Eclipse-Projekt für die Metamodellierung `http://www.eclipse.org/modeling/emf/` als Teil des Eclipse Modeling Project `http://www.eclipse.org/modeling/`
Ecore

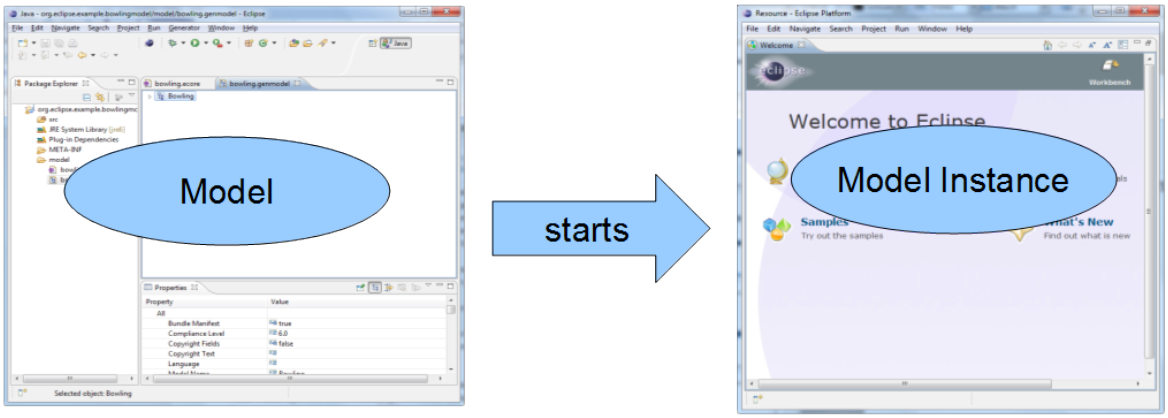
- Kern von EMF
- Implementierung von OMGs Essential MOF (EMOF)
- EClass : represents a class, with zero or more attributes and zero or more references.
- EAttribute : represents an attribute which has a name and a type.
- EReference : represents one end of an association between two classes. It has flag to indicate if it represent a containment and a reference class to which it points.
- EDataType : represents the type of an attribute, e.g. int, float or java.util.Date

Bildquelle: `http://eclipsesource.com/blogs/2011/03/22/what-every-eclipse-developer-should-know-about-emf-part-1/`

Eclipse Modeling Framework (EMF)



Eclipse Modeling Framework (EMF)



Werkzeuge in EMF

- Generierung von Java-Code aus Ecore-Metamodelle
- Generierung von Java-Code zur Bearbeitung von Metamodellen
- Serialisierung von Metamodellen in XMI (basiert auf XML)
- Baumeditor zur direkten Modellierung von Metamodellen und Modellen
- UML-Klassendiagrammartiger grafischer Editor

Graphical Modeling Framework (GMF)

- Eclipse-Projekt zur modellgetriebenen Erstellung grafischer Editoren für Ecore-Modelle
- Teil des Eclipse Modeling Project

Editorbau mit GMF

- Beschreibung von Modellen für verschiedene Aspekte des Editors
- Besonders geeignet für die schnelle Erstellung einfacher grafischer Editoren
- Fortgeschrittene Editoren erfordern Änderungen am Quellcode und an GMF-Templates

http://www.eclipse.org/modeling/

Textuelle Modellierung mit Xtext

Xtext – Language Development Framework

- Eclipse-Projekt zur Entwicklung externer textueller DSLs basierend auf EMF
- Teil des Eclipse Modeling Project
- Definition von EBNF-artigen Grammatiken, die abstrakte und konkrete Syntax gleichzeitig darstellen
- Verwendung von Xpand-Templates zur Code-Generierung

```
grammar org.example.domainmodel.Domainmodel
    with org.eclipse.xtext.common.Terminals

generate domainmodel
   "http://www.example.org/domainmodel/Domainmodel"

Model:
    greetings+=Greeting*;

Greeting:
    'Hello' name=ID '!';
```


http://www.eclipse.org/Xtext/

Modelltransformationen

Definition

Modelltransformationen: Abbildung einer Menge von Modellen auf eine andere Menge von Modellen

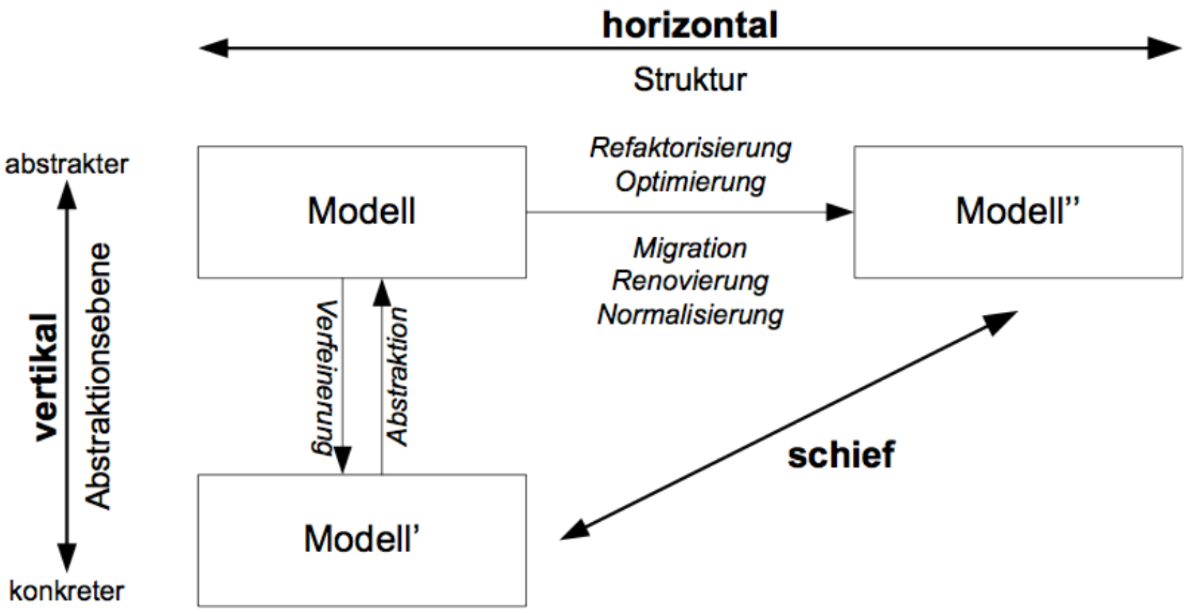
Varianten:

- Modell-zu-Modell-Transformationen (M2M, Mappings)
- Modell-zu-Text-Transformationen (M2T, Templates)

Modelltransformationen werden in der Regel

- auf Metamodellen beschrieben und
- auf Modellinstanzen angewendet

Arten von Modelltransformationen



– Reussner und Hasselbring (2009)

Modell-zu-Modell-Transformationen

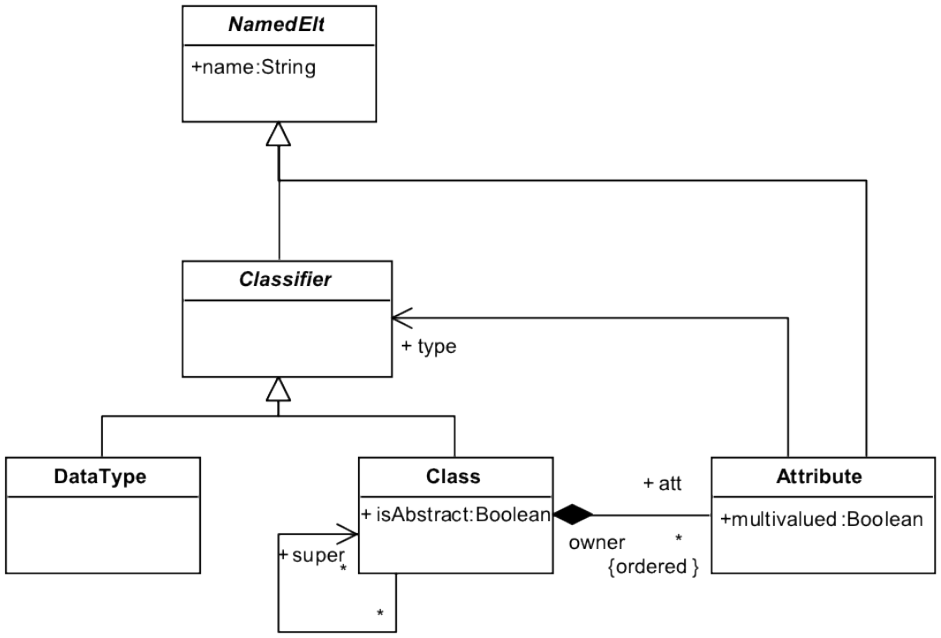
Erstellung von Modellen eines anderen Blickwinkels

- Überführen von Modellen höherer Abstraktionsebene in niedere Abstraktionsebenen
- Verfeinerung von Modellen

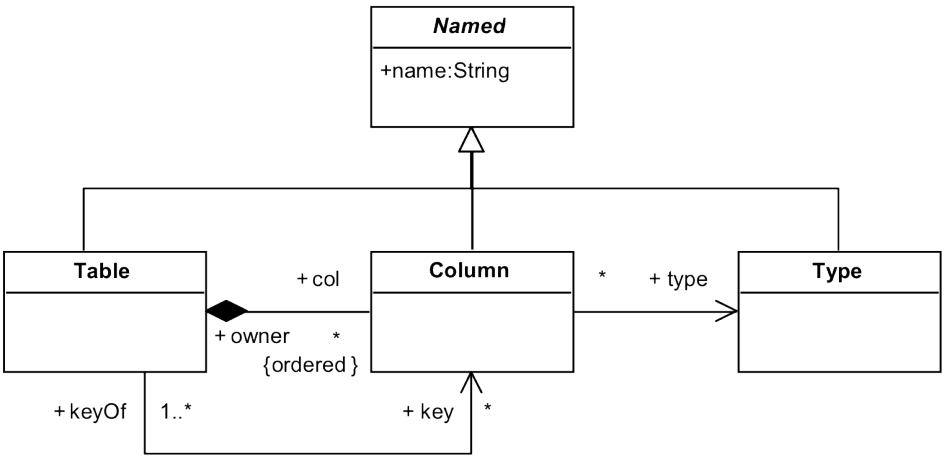
M2M-Transformationssprachen

- Query View Transformation (QVT) Operational Mapping
- Query View Transformation (QVT) Relations
- Atlas Transformation Language (ATL)
- Xtend aus dem (Eclipse) Xtext Language Development Framework

ATL-Beispiel: Metamodell für Quelle



ATL-Beispiel: Metamodell für Ziel



ATL-Beispiel: Transformation (Header)

```
module Class2Relational;
create OUT : Relational from IN : Class;
uses strings;
-- inheritance is not supported yet

helper def: objectIdType : Relational!Type =
Class!DataType.allInstances()
->select(e | e.name = 'Integer')->first();
```


ATL-Beispiel: Transformation

```
rule DataType2Type {
from
    dt : Class!DataType
to
    out : Relational!Type (
        name <- dt.name
    )
}
```

For each DataType instance, a Type instance has to be created. Their names have to correspond.

ATL-Beispiel: Transformation

```
rule Class2Table {
from
    c : Class!Class
to
    out : Relational!Table (
        name <- c.name,
        -- Columns are generated from Attributes in another
        --  rule not explicitly called here !
        col <- Sequence {key}->
            union(c.attr->select(e | not e.multiValued)),
        key <- Set {key}
    ),
    key : Relational!Column (
      name <- 'objectId',
      type <- thisModule.objectIdType
    )
}
```

For each Class instance, a Table instance has to be created.

- Their names have to correspond.
- The col reference set has to contain all Columns that have been created for single- valued attributes and also the key described in the following.
- The key reference set has to contain a pointer to the key described in the following.
- An Attribute instance has to be created as key
  - Its name has to be set to 'objectId'
  - Its type reference has to reference a Type with the name Integer which - if not yet existing - has to be created.

ATL-Beispiel: Transformation

```
rule SingleValuedDataTypeAttribute2Column {
from
  a : Class!Attribute (
    a.type.oclIsKindOf(Class!DataType) and not a.multiValued
  )
to
  out : Relational!Column (
   name <- a.name,
   type <- a.type
  )
}
```

For each single-valued Attribute of the type Class, a new Column has to be created.

- Its name has to be set to the attribute's name concatenated with 'id'.
- Its type reference has to reference a Type with the name Integer which - if not yet existing - has to be created.

Nicht gezeigt: Regeln für multivariate Attribute. Siehe `http://www.eclipse.org/m2m/atl/atlTransformations/Class2Relational/ExampleClass2Relational%5Bv00.01%5D.pdf`

Modell-zu-Text-Transformationen

Arten generierten Texts aus Quellmodellen:

- Programmcode
- Konfigurationsdateien
- Dokumentation wie z.B. Javadoc

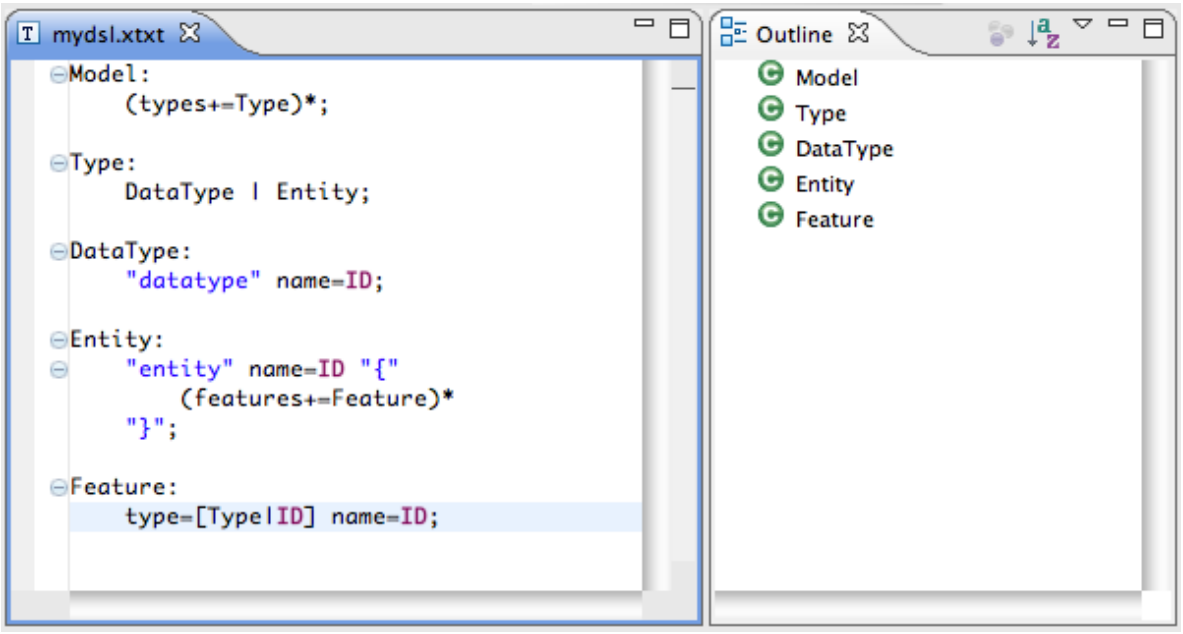
Varianten von Modell-zu-Text-Transformationen:

- Visitor-basiert
- Template-basiert

Template-basierte Werkzeuge:

- Xpand aus dem (Eclipse) Xtext Language Modeling Framework
- AndroMDA
- Java Emitter Templates (JET)

Xpand (Xtext-Grammatik)



Bildquelle:

`http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/xtext_tutorial.html`

Xpand (Template)

Main.xpt

```
«IMPORT mydsl»

«DEFINE main FOR Model»
   «EXPAND javaBean FOREACH types.typeSelect(Entity)»
«ENDDEFINE»

«DEFINE javaBean FOR Entity»
«FILE name+".java"»

public class «name» {
    «FOREACH features AS f»
    private «f.type.name» «f.name»;

    public «f.type.name» get«f.name.toFirstUpper()»() {
        return «f.name»;
    }

    public void set«f.name.toFirstUpper()»(«f.type.name» «f.name») {
        this.«f.name» = «f.name»;
    }
    «ENDFOREACH»
}
«ENDFILE»
«ENDDEFINE»
```


Xpand-Tutorial: `http://www.peterfriese.de/getting-started-with-code-generation-with-xpand/`

Bildquelle:

`http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/xtext_tutorial.html`

Vor- und Nachteile von DSLs

Code-Generierung aus Modellen

+ Arbeitsersparnis bei regulären und wohl verstandenen Domänen
- wenigstens eine Referenzimplementierung notwendig
- generierter Code muss mit handgeschriebenem integriert werden
- generierter Code sollte *readonly* sein

– Code-Generatoren müssen erstellt und gewartet werden

Analysefähigkeit

+ abstraktere Darstellung

– „der Generator ist die Spezifikation“

– eigene Analysewerkzeuge notwendig

– eigene Debugging-Werkzeuge notwendig

Wiederholungs- und Vertiefungsfragen

- Was ist die Kernidee der modellgetriebenen Entwicklung? Welche Vorteile verspricht man sich davon?
- Was sind die Merkmale eines Modells im Allgemeinen?
- Was sind Metamodelle und Metametamodelle? Wozu werden sie benötigt?
- Welche Aspekte sind bei der Definition einer Sprache festzulegen?
- Was ist eine domänenspezifische Sprache (DSL)? Was sind die Unterschiede zu einer herkömmlichen Programmiersprache?
- Welche Arten von DSLs unterscheidet man?
- Beschreiben Sie die Unterstützung von EMF für die modellgetriebene Softwareentwicklung.
- Was sind Modelltransformationen und welche Arten gibt es?
- Erläutern Sie konzeptionell Modell-zu-Modell- sowie Model-zu-Text-Transformationen?
- Wie können insbesondere Model-zu-Text-Transformationen realisiert werden?

Demo zur Modellgetriebenen Softwareentwicklung
Demo von EMF und XPand
Modell-zu-Modell-Transformation mit QVT

Demo von EMF und XPand

Schritte:

1. install plug-ins
2. create reference implementation
3. create generator/modeling project
4. create meta-model
5. create model instance
6. create code generator
7. run code generator

Notwendige Plug-Ins

Eclipse/JDT

- EMF - Eclipse Modeling Framework SDK
- Eclipse Plug-In Development Environment
- Ecore Tools SDK
- XPand SDK
- MWE SDK - Model Flow Engine

Für Modell-zu-Modell-Transformationen noch zusätzlich:

- Operational QVT SDK

reference implementation I

```
1  /**
2   * A state machine.
3   *
4   * Transition table:
5   *
6   *    state   | event   | successor state
7   *   ---------------------------------
8   *    Start   | run     | Idle
9   *    Idle    | wakeup  | Busy
10  *    Idle    | finish  | Final
11  *    Busy    | done    | Idle
12  *
13  *    All other events not listed in the
14  *    table are illegal events.
15  */
16
17 package statecharts;
18
19 import statecharts.UnexpectedEvent;
20
```

reference implementation II

```
21 public class StateMachine {
22
23   enum Event {run, done, wakeup, finish};
24
25   enum State {Start, Final, Busy, Idle};
26
27   private State state = State.Start;
28
29   public State getState() {
30     return state;
31   }
32
33
34
35
36
37
38
39
40
41   public void signal(Event event) throws UnexpectedEvent {
```


reference implementation III

```
42     switch(state) {
43     case Start:
44       switch(event) {
45       case run:
46         state = State.Idle;
47         break;
48       default:
49         error();
50         break;
51       }
52       break;
53     case Final:
54       switch(event) {
55       default:
56         error();
57         break;
58       }
59       break;
60     case Busy:
61       switch(event) {
62       case done:
```

reference implementation IV

```
63            state = State.Idle;
64            break;
65          default:
66            error();
67            break;
68          }
69          break;
70        case Idle:
71          switch(event) {
72          case wakeup:
73            state = State.Busy;
74            break;
75          case finish:
76            state = State.Final;
77            break;
78          default:
79            error();
80            break;
81          }
82          break;
83        }
```


reference implementation V

```
84    }
85
86    private void error() throws UnexpectedEvent {
87      System.err.println("Error");
88      throw new UnexpectedEvent();
89    }
90  }
```

test of reference implementation I

```
1  package statecharts;
2
3  import static org.junit.Assert.assertEquals;
4
5  import org.junit.Test;
6
7  import statecharts.UnexpectedEvent;
8
9  public class StateMachineTest {
10
11   /**
12    * Applies the sequence of events to s
13    * @param s the state machine for which the events should be si
14    * @param events the sequence of events to be signaled
15    * @throws UnexpectedEvent
16    */
17   private void apply(StateMachine s, StateMachine.Event[] events)
18     for(StateMachine.Event e : events) {
19       s.signal(e);
20     }
```


test of reference implementation II

```
21   }
22
23   /**
24    * @return a new state machine to be tested
25    */
26   private StateMachine newStateMachine() {
27     return new StateMachine();
28   }
29
30   @Test
31   public void testInit() throws UnexpectedEvent {
32     StateMachine s = newStateMachine();
33     assertEquals(StateMachine.State.Start, s.getState());
34   }
35
36   @Test
37   public void testIdle() throws UnexpectedEvent {
38     StateMachine s = newStateMachine();
39     StateMachine.Event[] events = {StateMachine.Event.run};
40     apply(s, events);
41     assertEquals(StateMachine.State.Idle, s.getState());
```

test of reference implementation III

```
42    }
43
44    @Test(expected=UnexpectedEvent.class)
45    public void testIdleIllegalDone() throws UnexpectedEvent {
46      StateMachine s = newStateMachine();
47      StateMachine.Event[] events = {StateMachine.Event.done};
48      apply(s, events);
49    }
50
51    @Test(expected=UnexpectedEvent.class)
52    public void testIdleIllegalFinish() throws UnexpectedEvent {
53      StateMachine s = newStateMachine();
54      StateMachine.Event[] events = {StateMachine.Event.finish};
55      apply(s, events);
56    }
57
58    @Test(expected=UnexpectedEvent.class)
59    public void testIdleIllegalWakeup() throws UnexpectedEvent {
60      StateMachine s = newStateMachine();
61      StateMachine.Event[] events = {StateMachine.Event.wakeup};
62      apply(s, events);
```


test of reference implementation IV

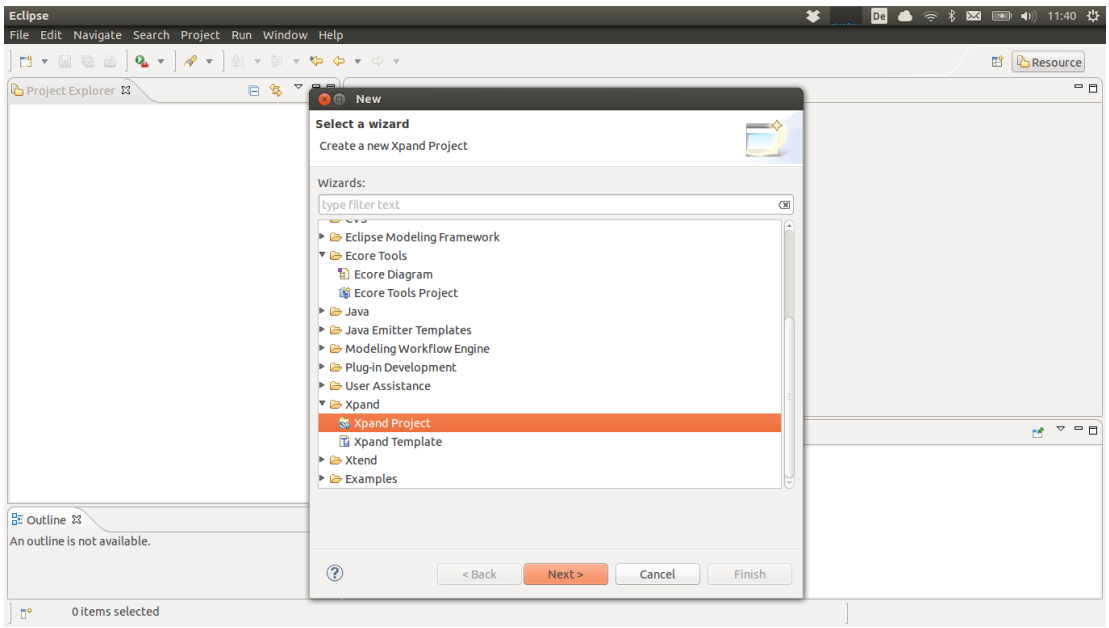
```
63    }
64
65    @Test
66    public void testBusy() throws UnexpectedEvent {
67      StateMachine s = newStateMachine();
68      StateMachine.Event[] events = {StateMachine.Event.run,
69                                     StateMachine.Event.wakeup};
70      apply(s, events);
71      assertEquals(StateMachine.State.Busy, s.getState());
72    }
73
74    @Test
75    public void testFinal() throws UnexpectedEvent {
76      StateMachine s = newStateMachine();
77      StateMachine.Event[] events = {StateMachine.Event.run,
78                        StateMachine.Event.wakeup,
79                        StateMachine.Event.done,
80                        StateMachine.Event.finish};
81      apply(s, events);
82      assertEquals(StateMachine.State.Final, s.getState());
83    }
```

test of reference implementation V

```
84 }
```

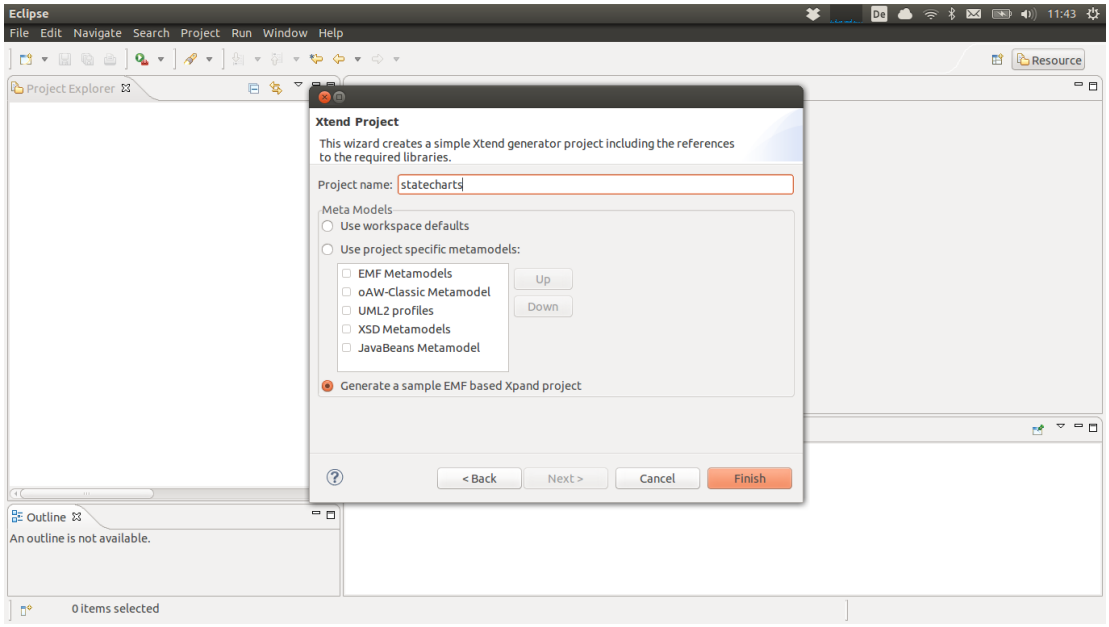

create generator/modeling project

Open the new project wizard and choose 'Xpand Project' from the list

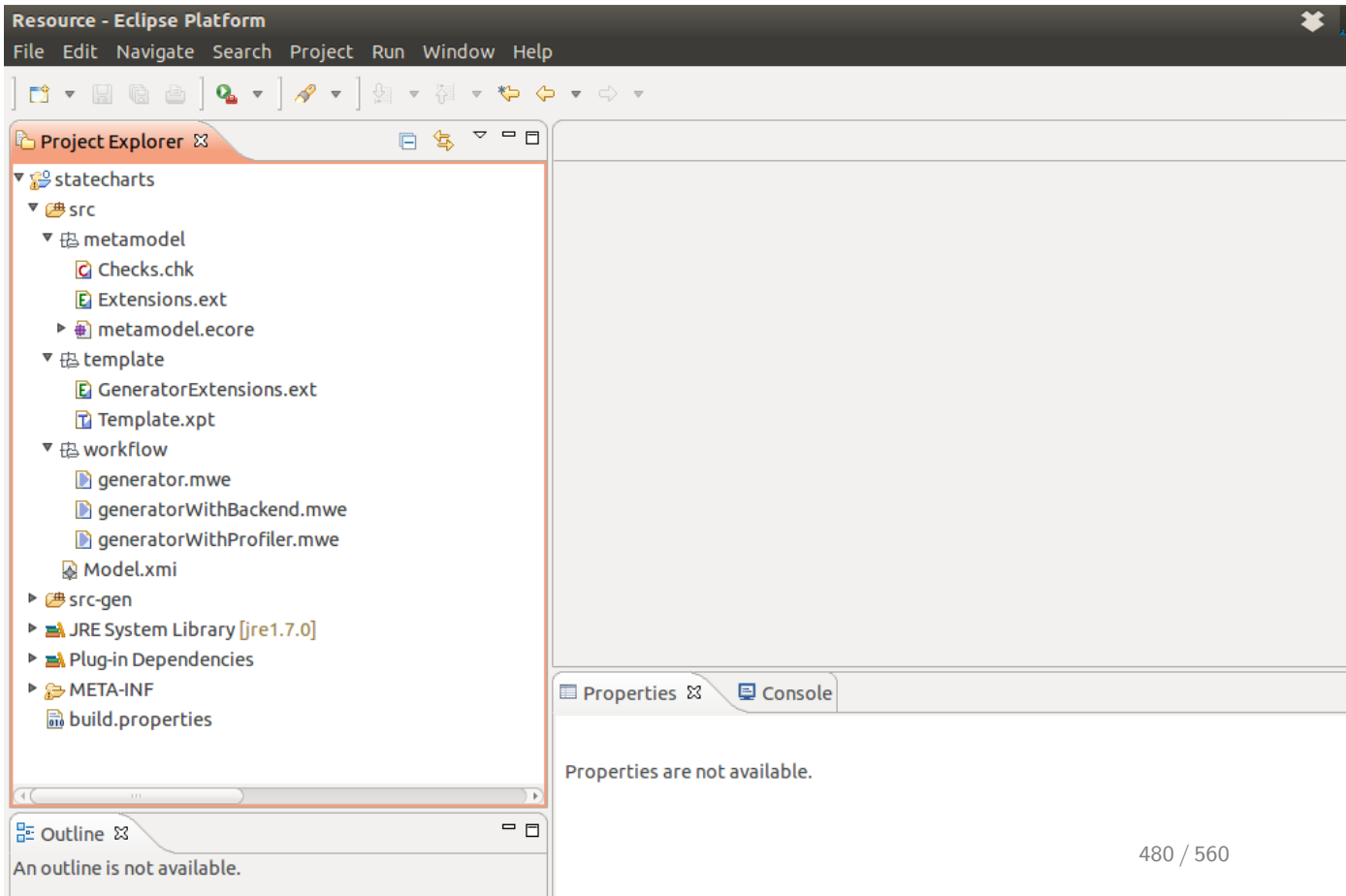


create generator/modeling project

1. Choose a name for your project (e.g., statecharts)
2. Select 'Create a sample EMF based Xpand project'
3. Press 'Finished'



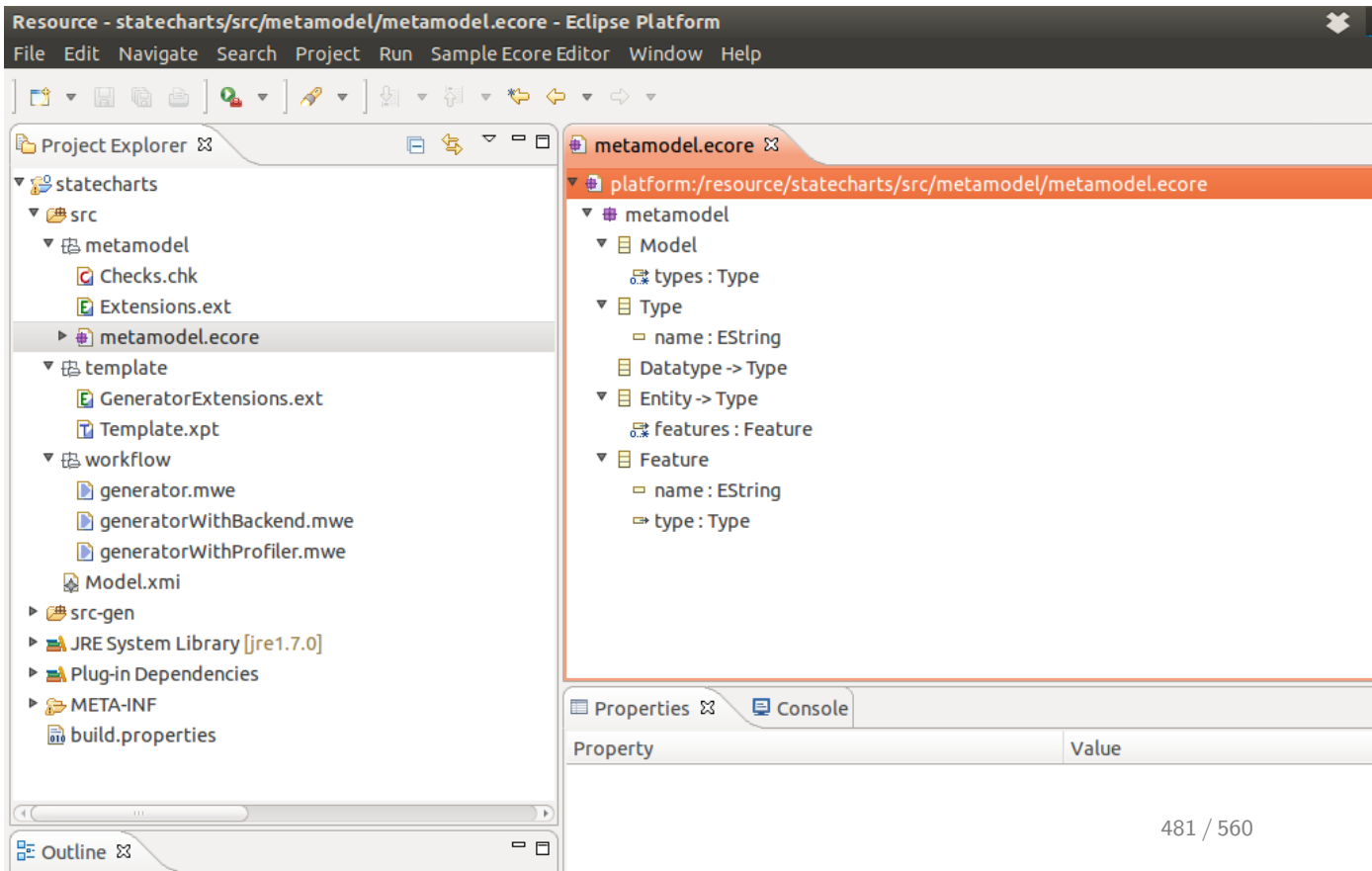
create generator/modeling project



- metamodel.ecore: Metamodell
- Template.xpt: Template für Code-Generierung
- generator.mwe: Workflow für Code-Generierung
- Model.xmi: Beispielmodell (Instanz des Metamodells)

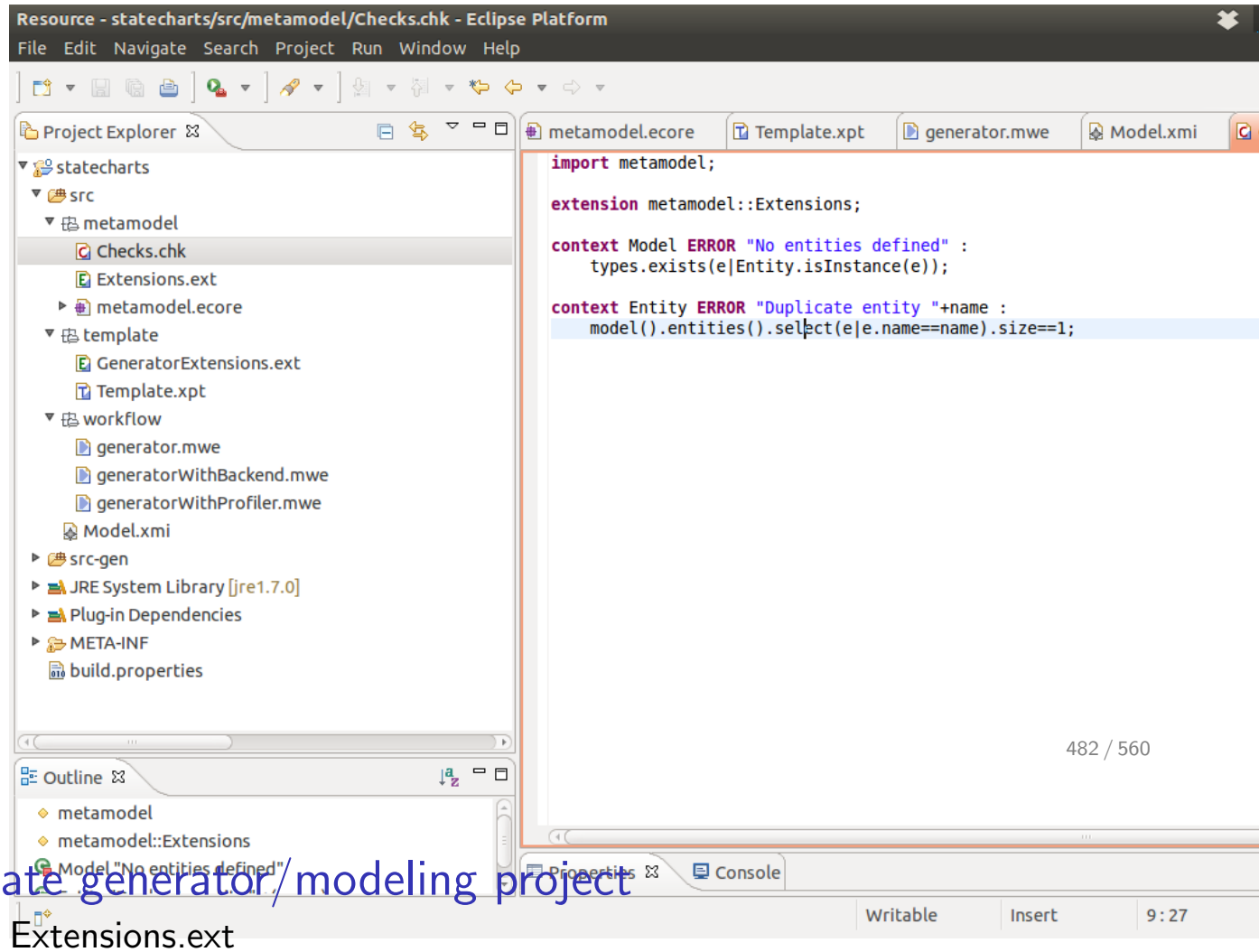
create generator/modeling project

Metamodell



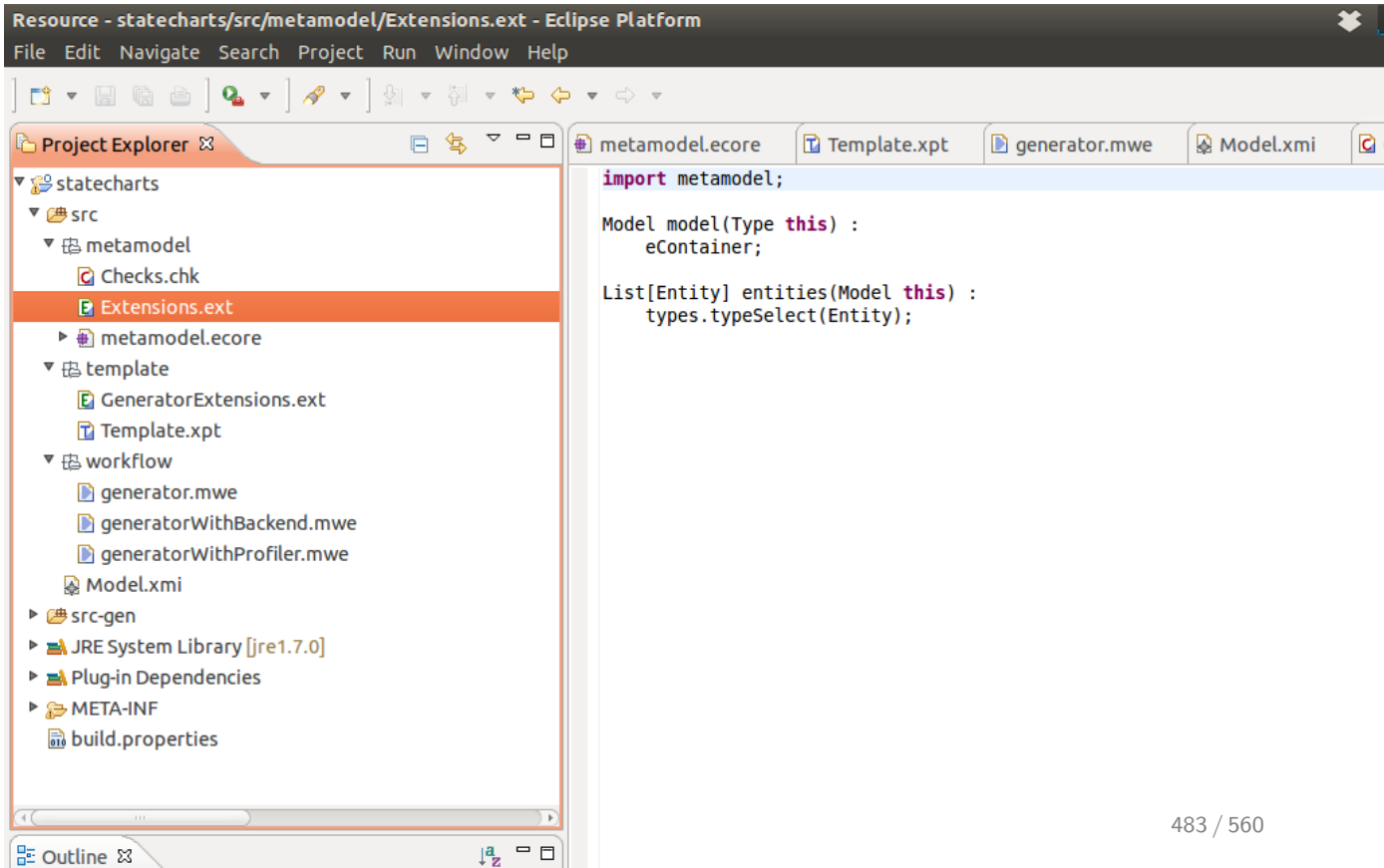
create generator/modeling project

Checks.chk



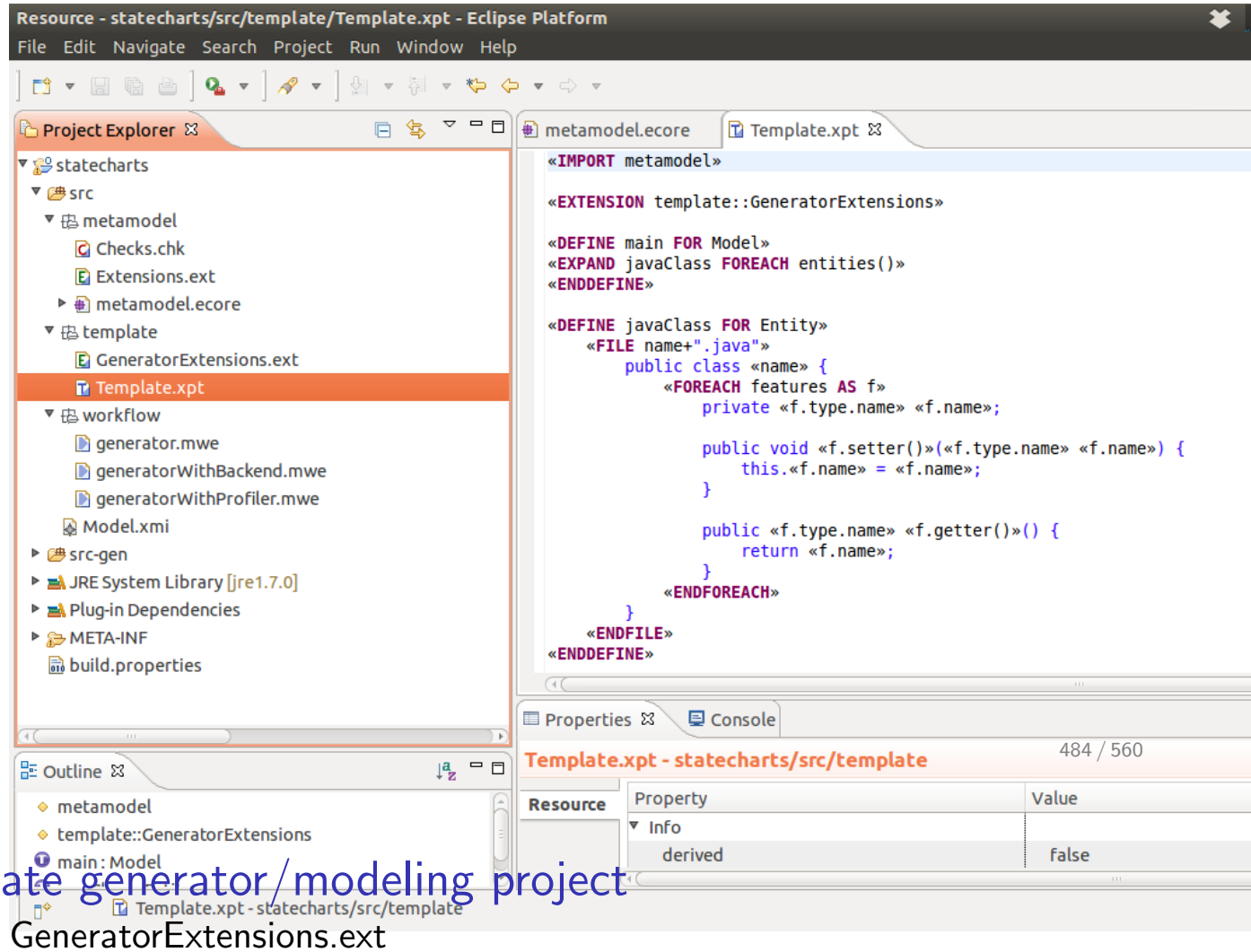
create generator/modeling project

Extensions.ext



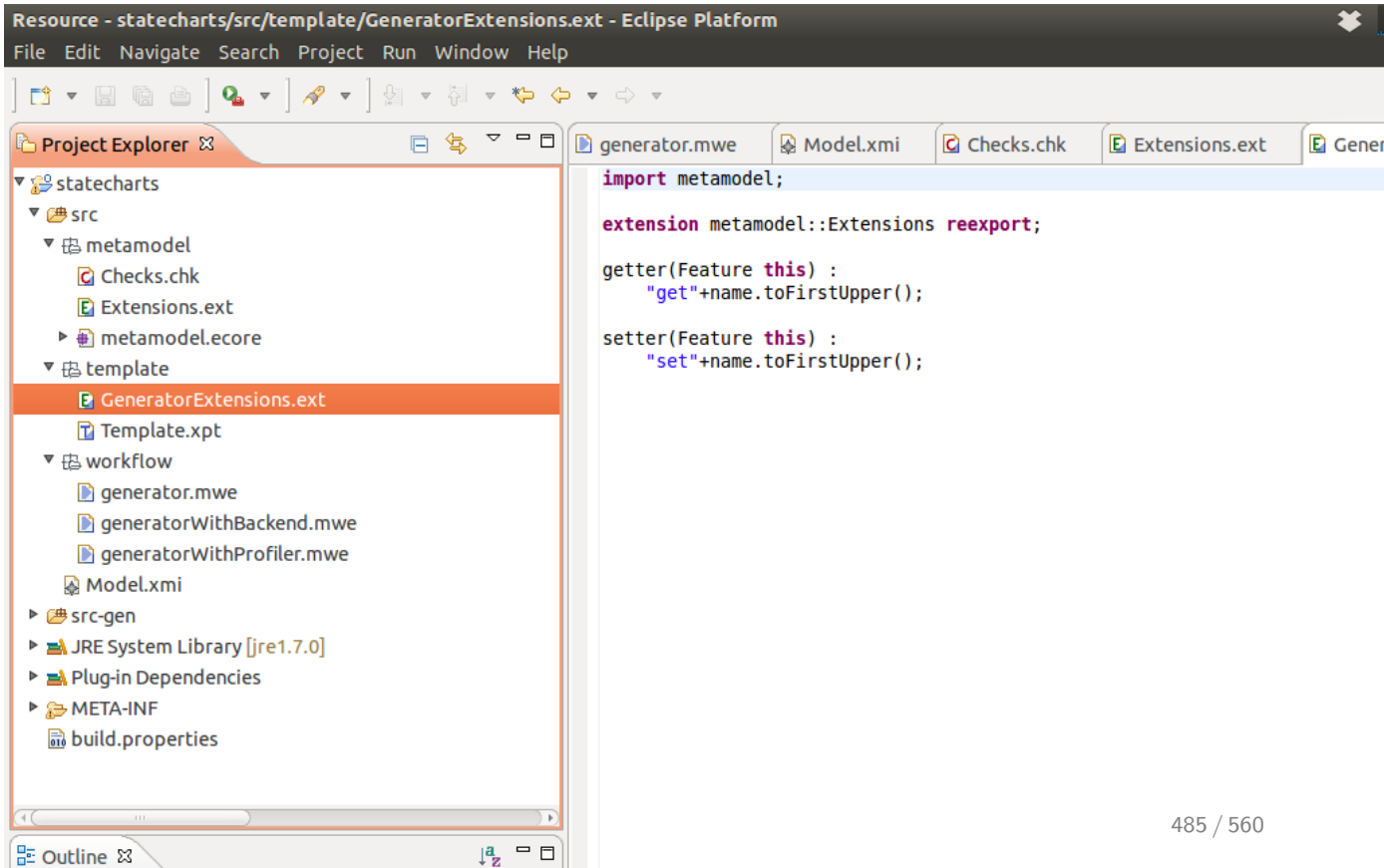
create generator/modeling project

Template für Code-Generierung



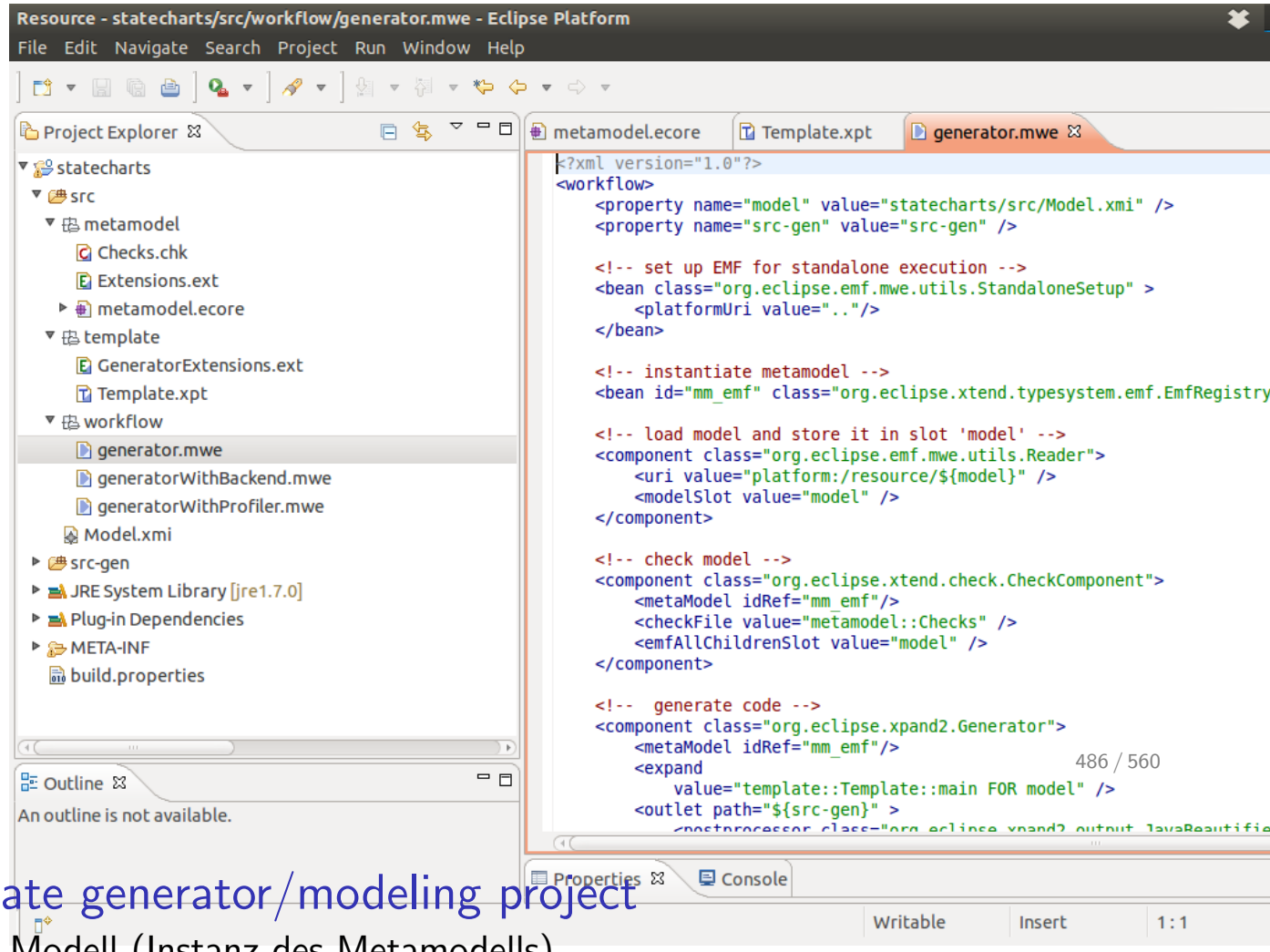
create generator/modeling project

GeneratorExtensions.ext



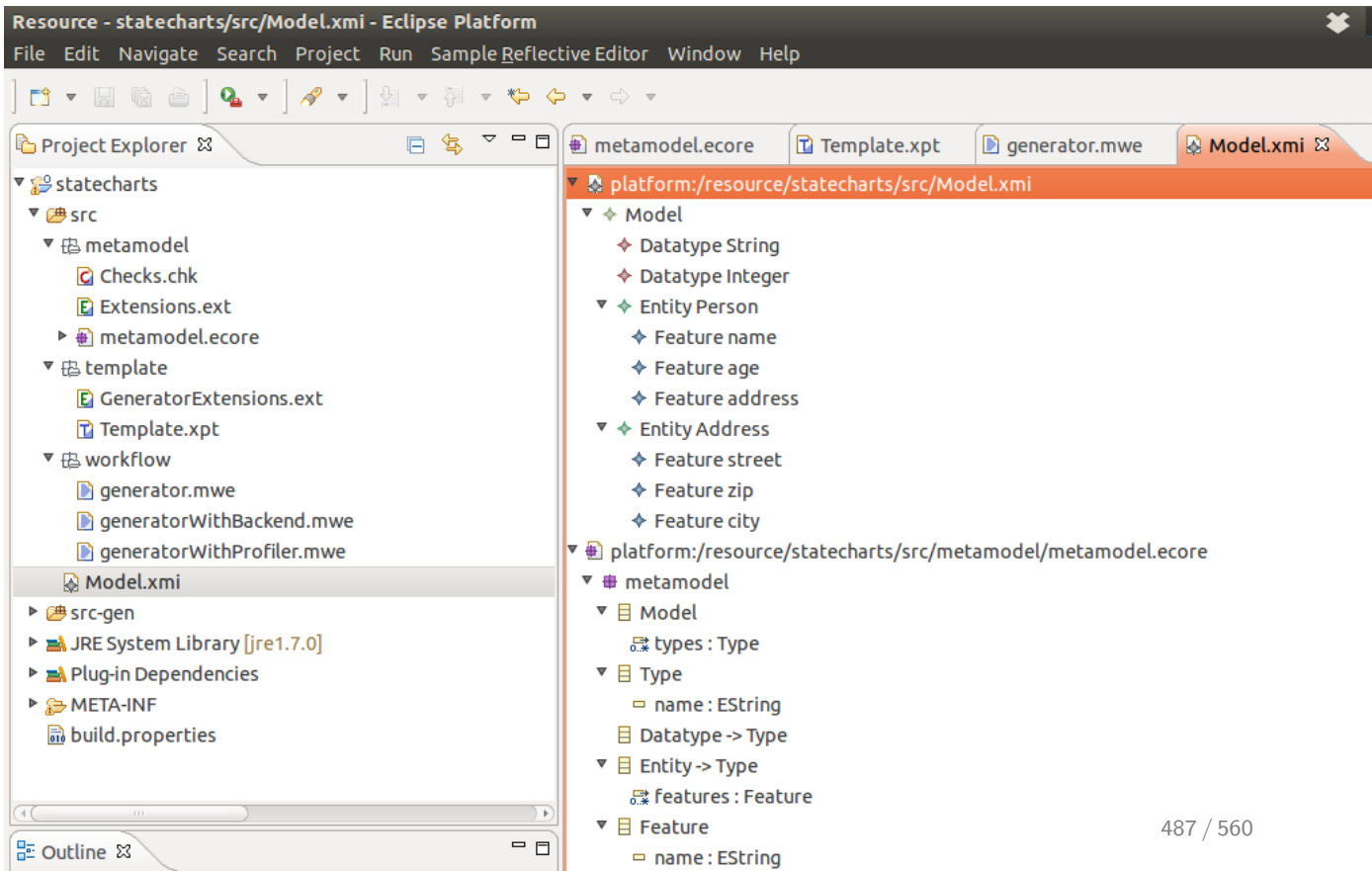
create generator/modeling project

Workflow für Code-Generierung



create generator/modeling project

Modell (Instanz des Metamodells)



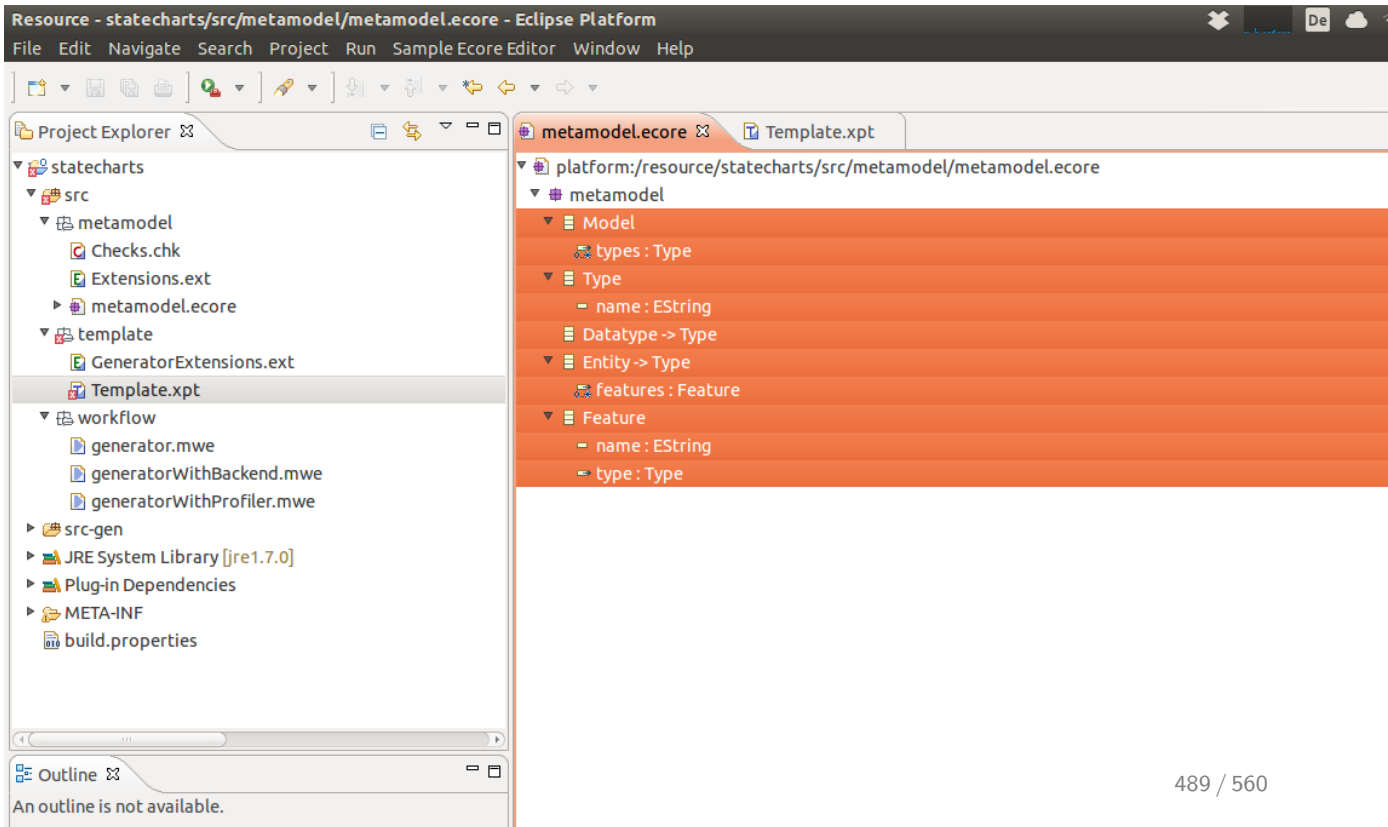
create generator/modeling project

Entferne alle spezifischen Deklarationen des generierten Beispiels

1. Open src/metamodel/Checks.chk and delete all its contents
2. Open src/metamodel/Extensions.chk and delete all its contents
3. Open src/template/GeneratorExtensions.ext and delete all its contents
4. Open src/template/Template.xpt and delete all its contents
5. Delete src/Model.xmi
6. Do not forget to save all modified files

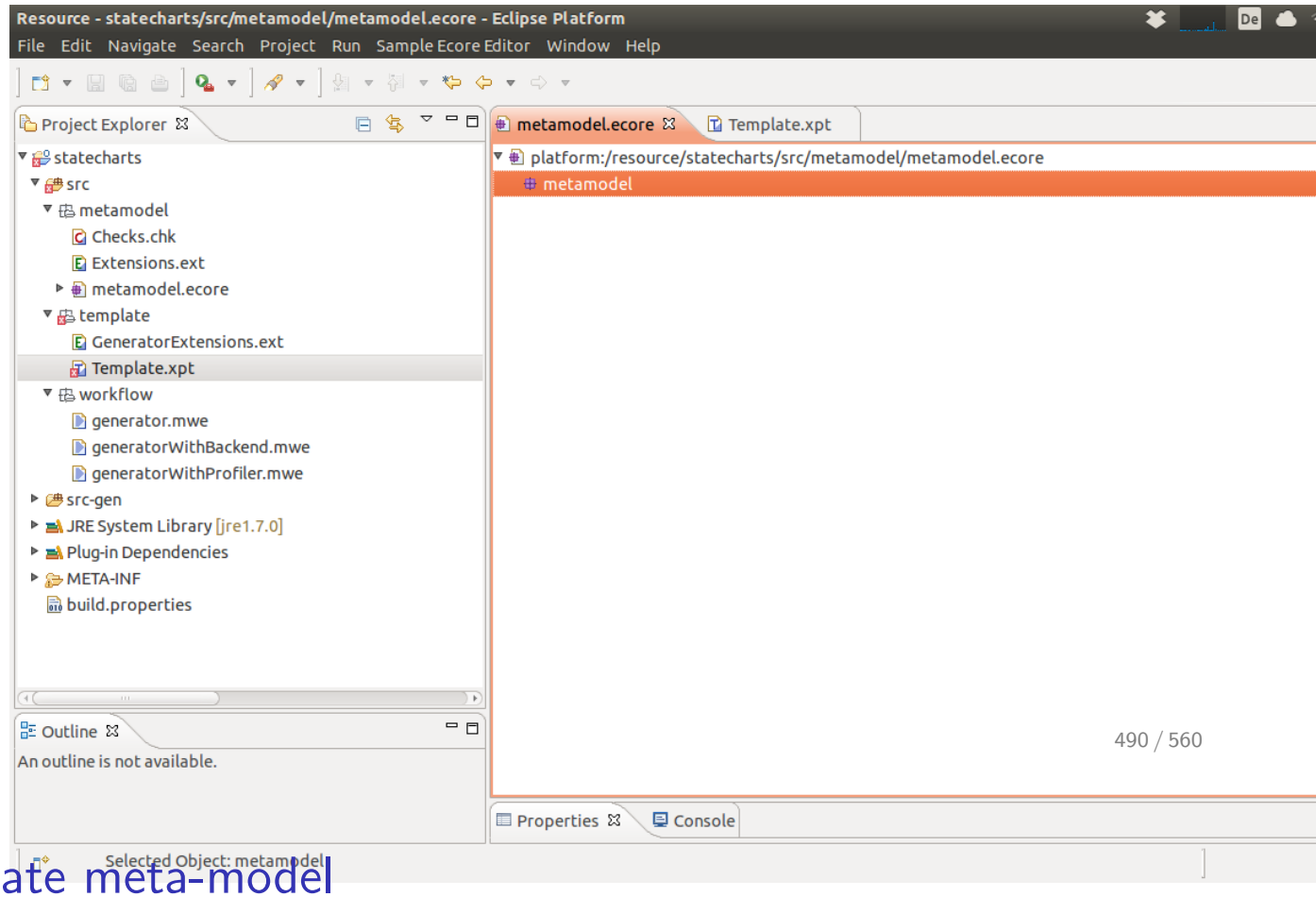
create generator/modeling project

Delete all children of metamodel in src/metamodel/metamodel.ecore



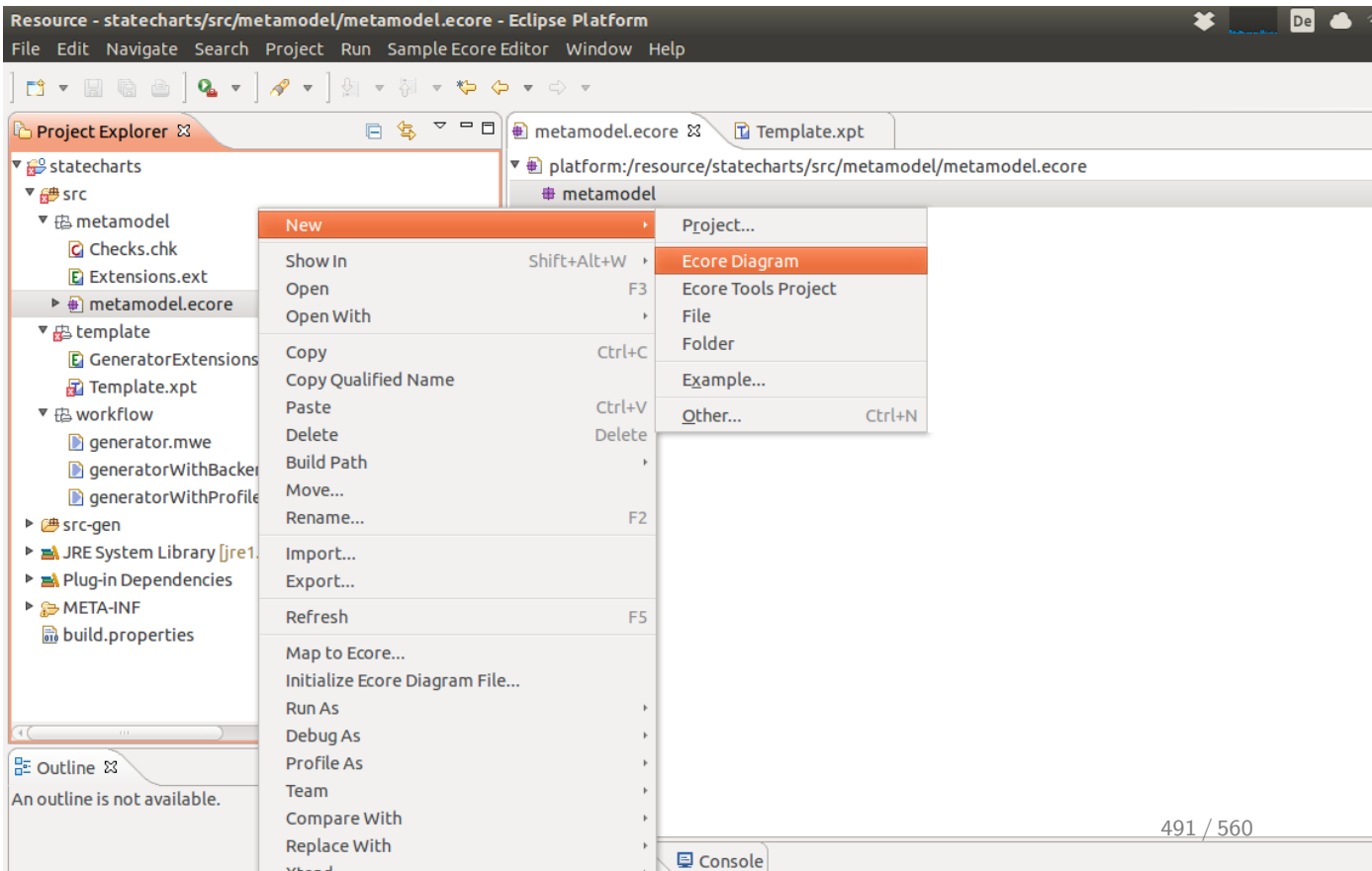
create generator/modeling project

Delete all children of metamodel in src/metamodel/metamodel.ecore



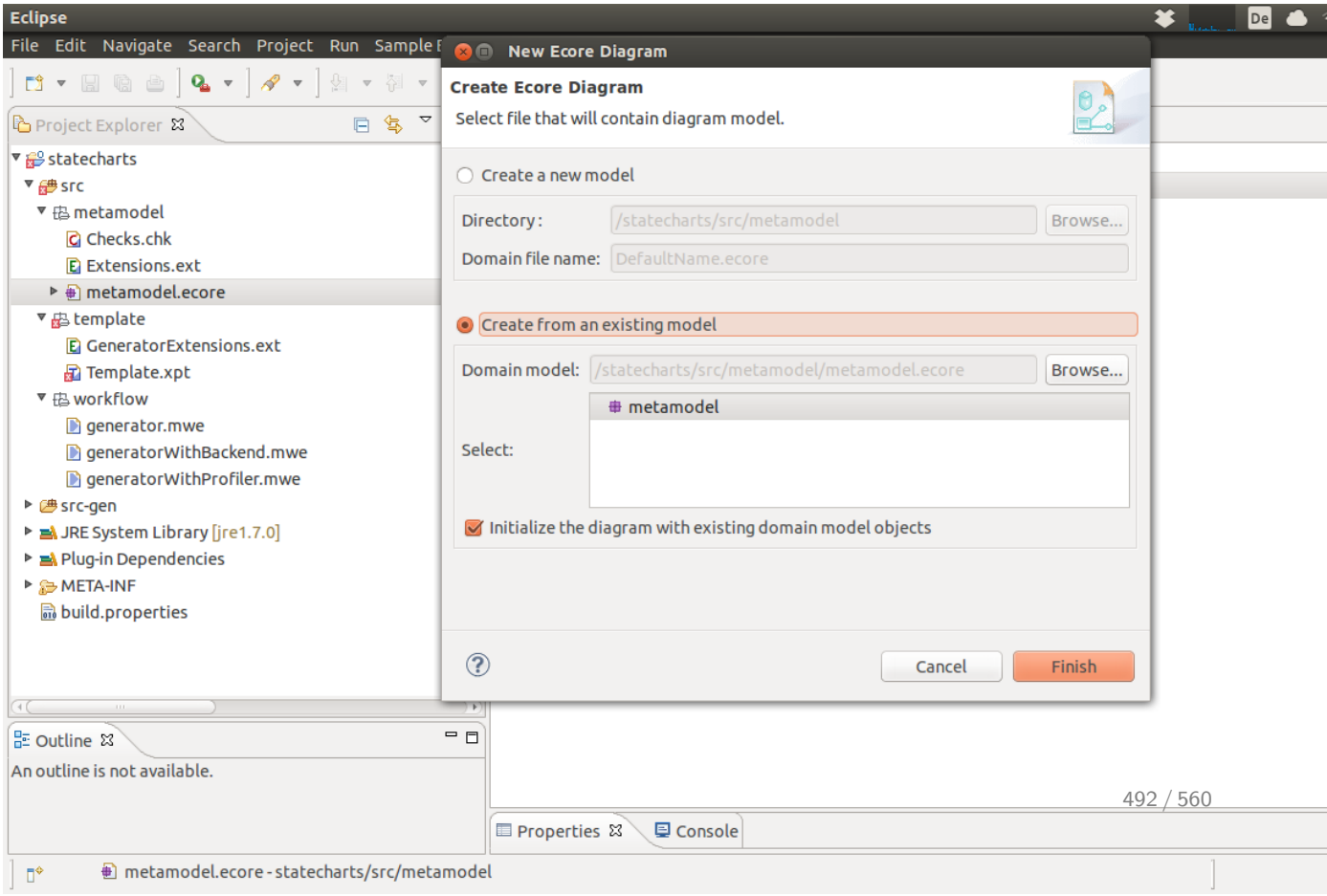
create meta-model

Open graphical meta-model editor



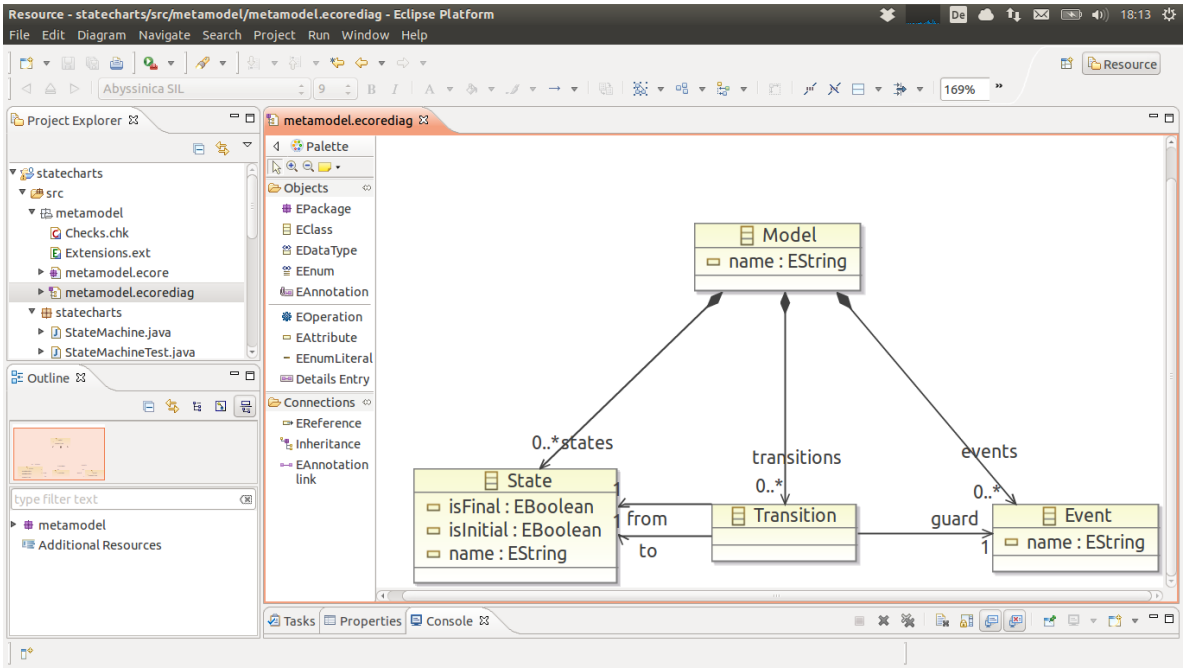
create meta-model

Press 'Finish'



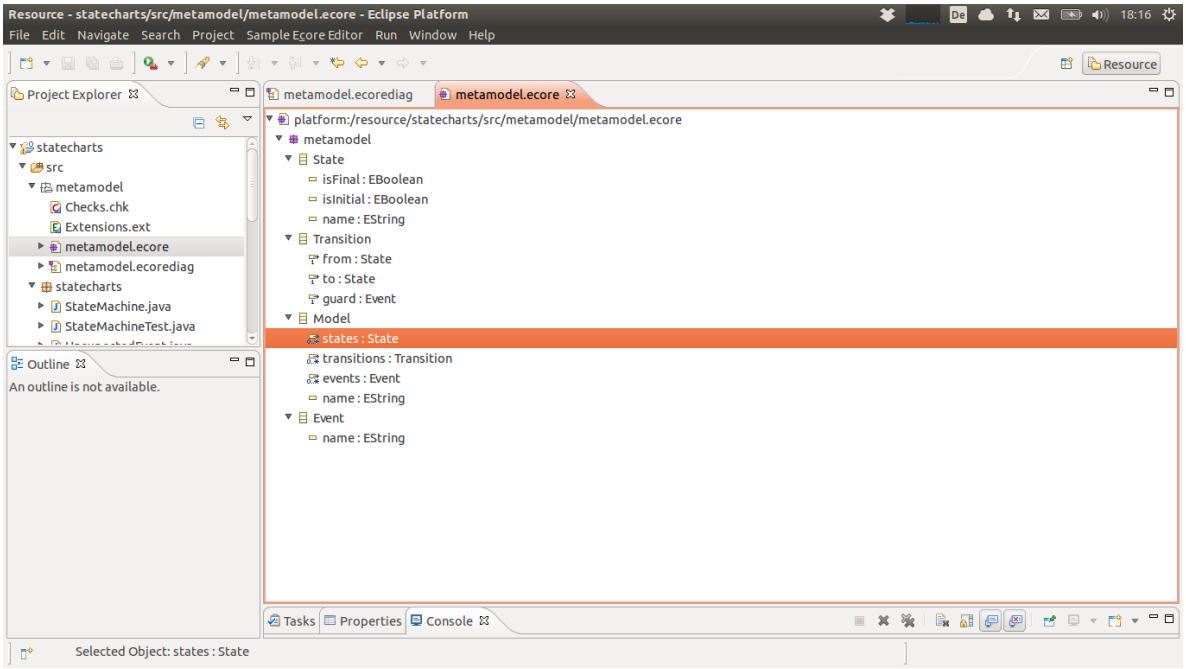
create meta-model

Create meta-model for state charts



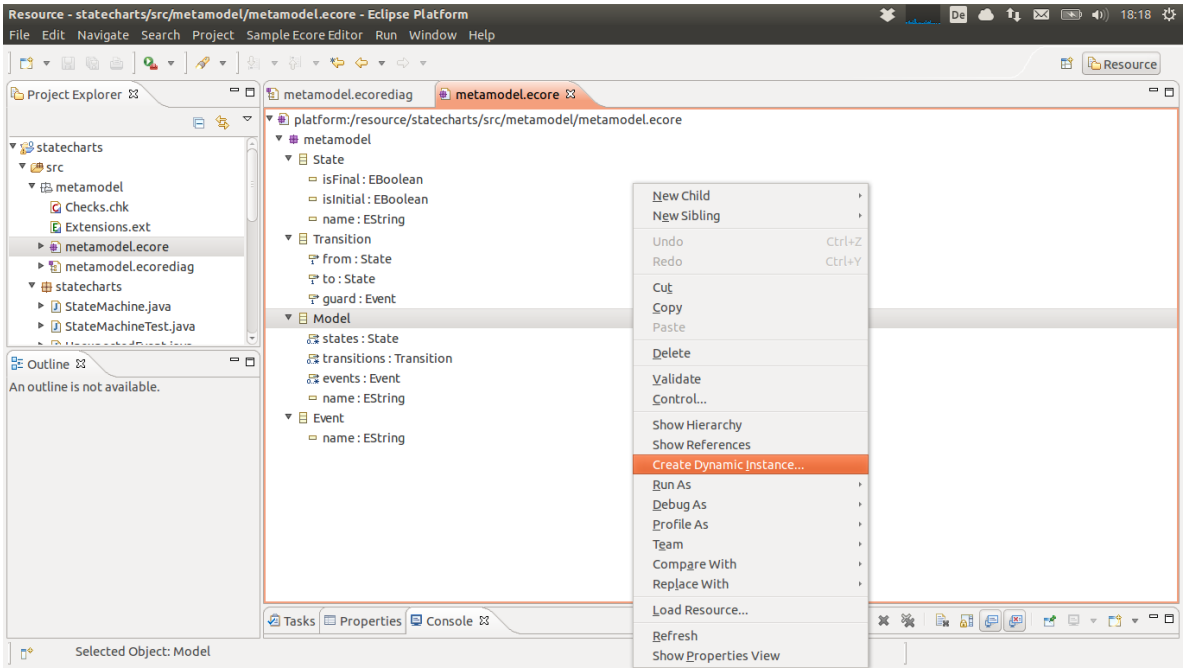
create meta-model

Model as a tree



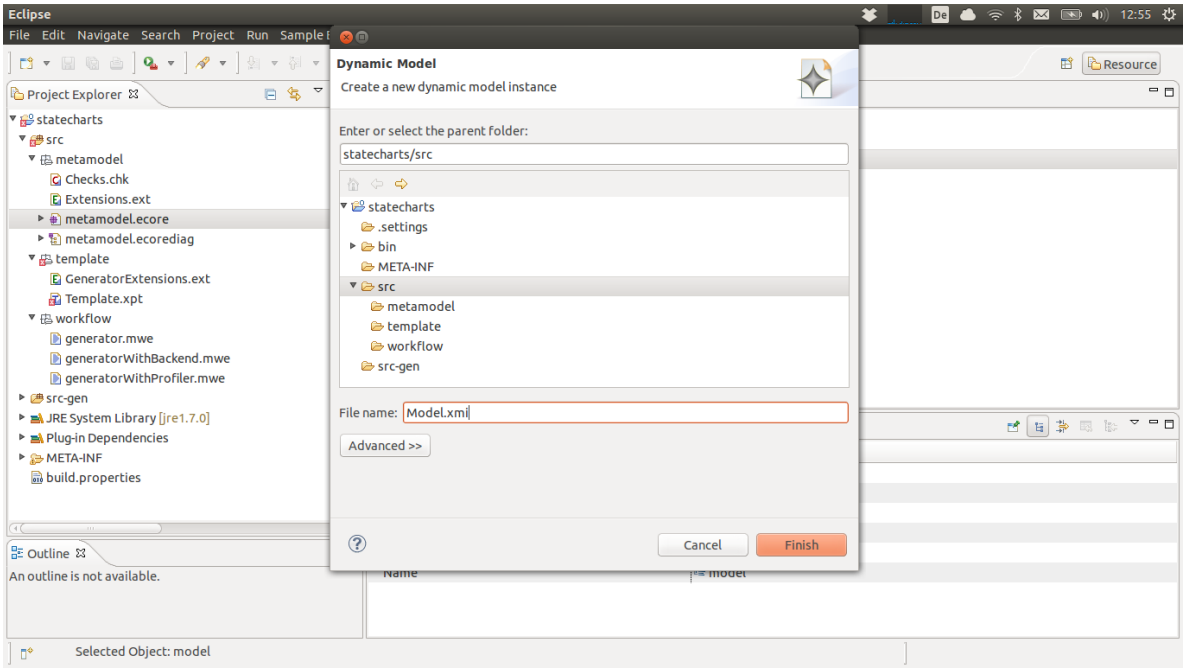
create model instance

Right-click *model* in *metamodel.ecore* and select *Create Dynamic Instance*



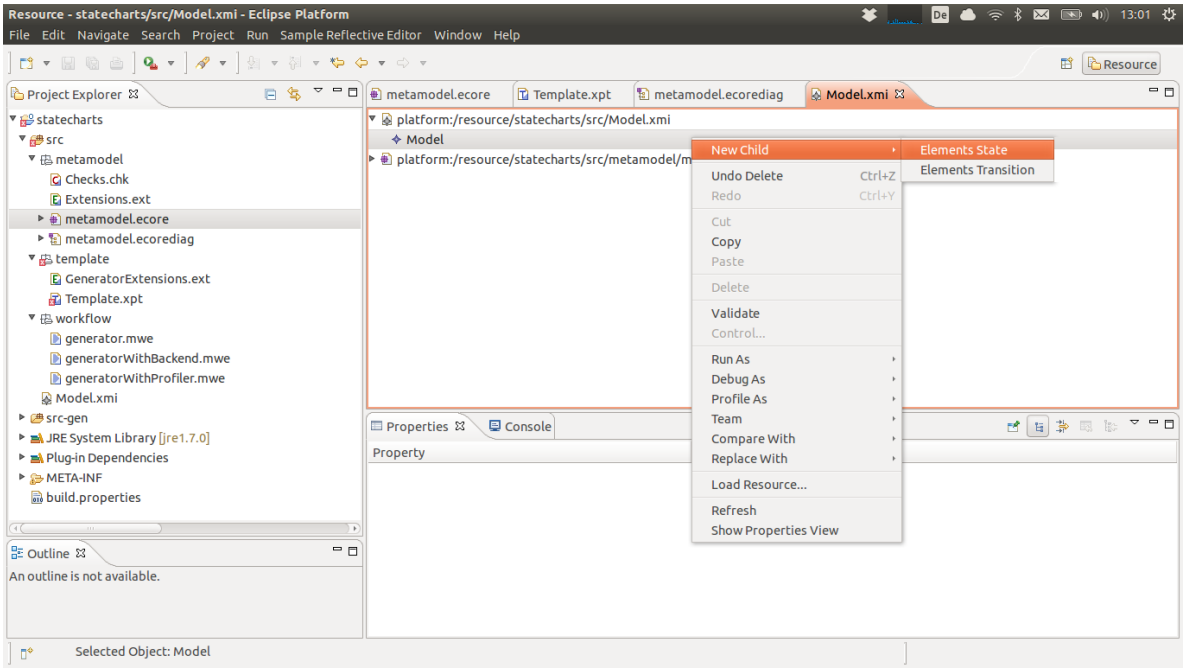
create model instance

Select parent folder `statecharts/src` and enter file name `Model.xmi`



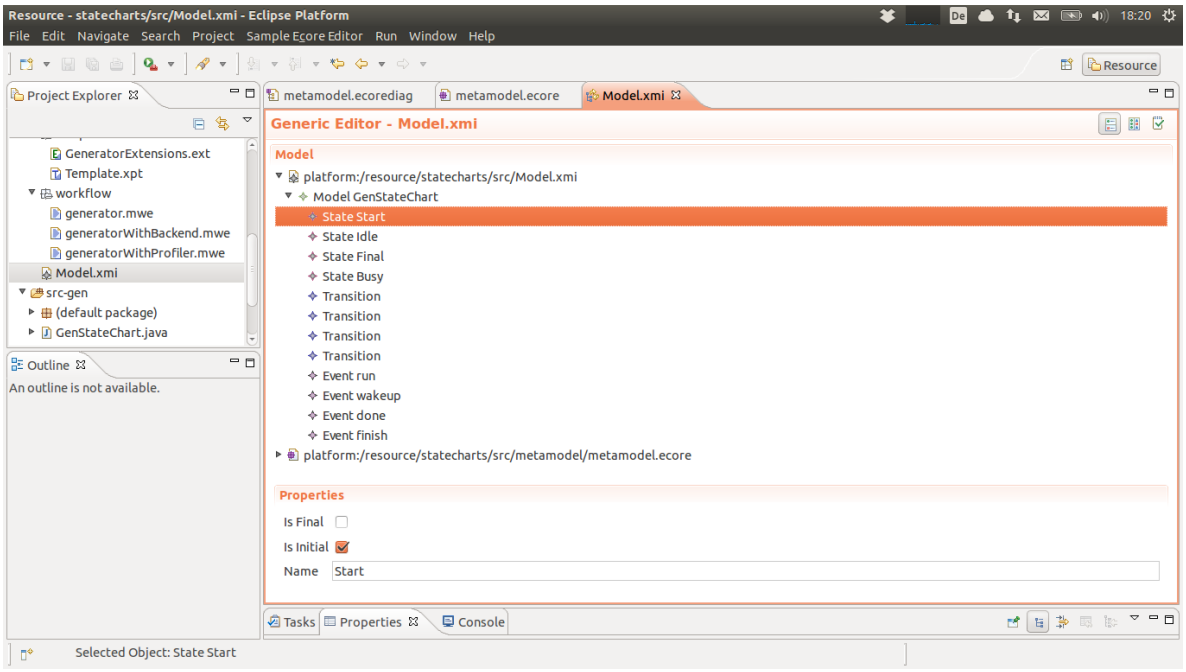
create model instance

Create new state



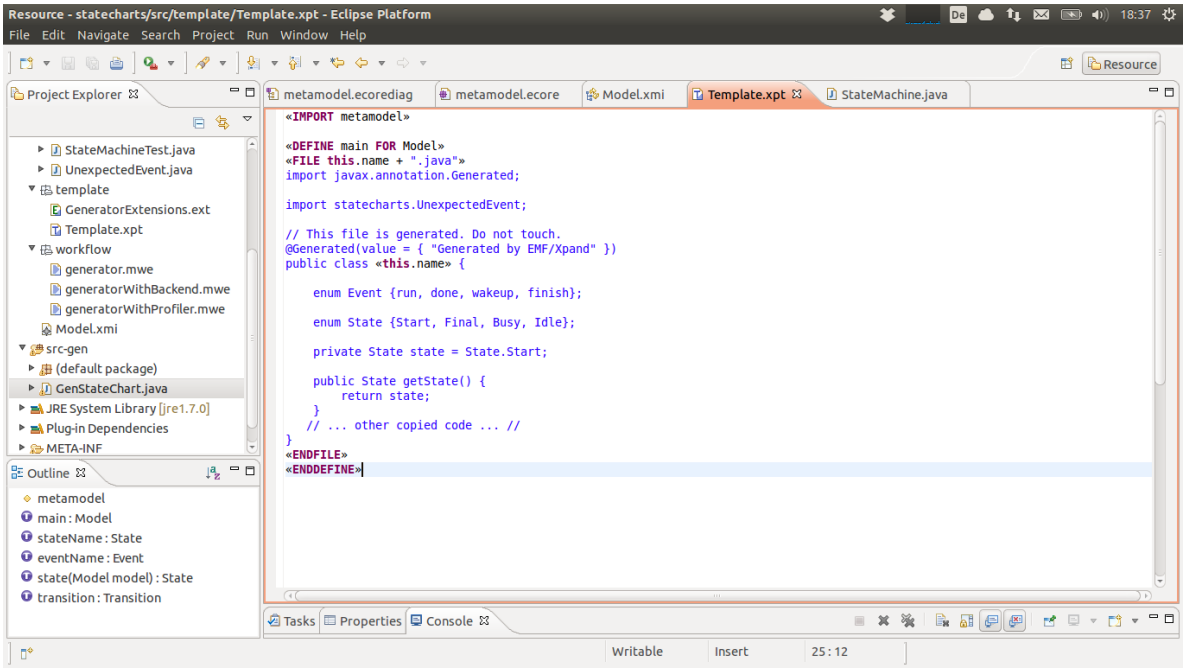
create model instance

Poplulate model with states and transitions



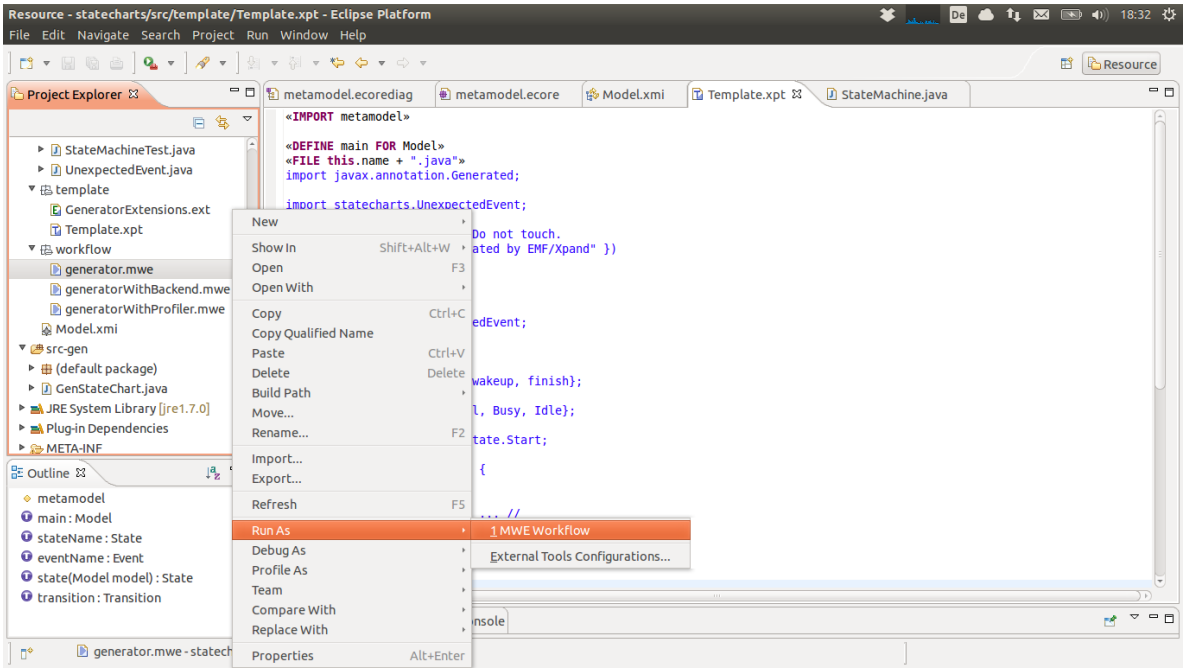
create code generator

Copy reference implementation into `Template.xpt`



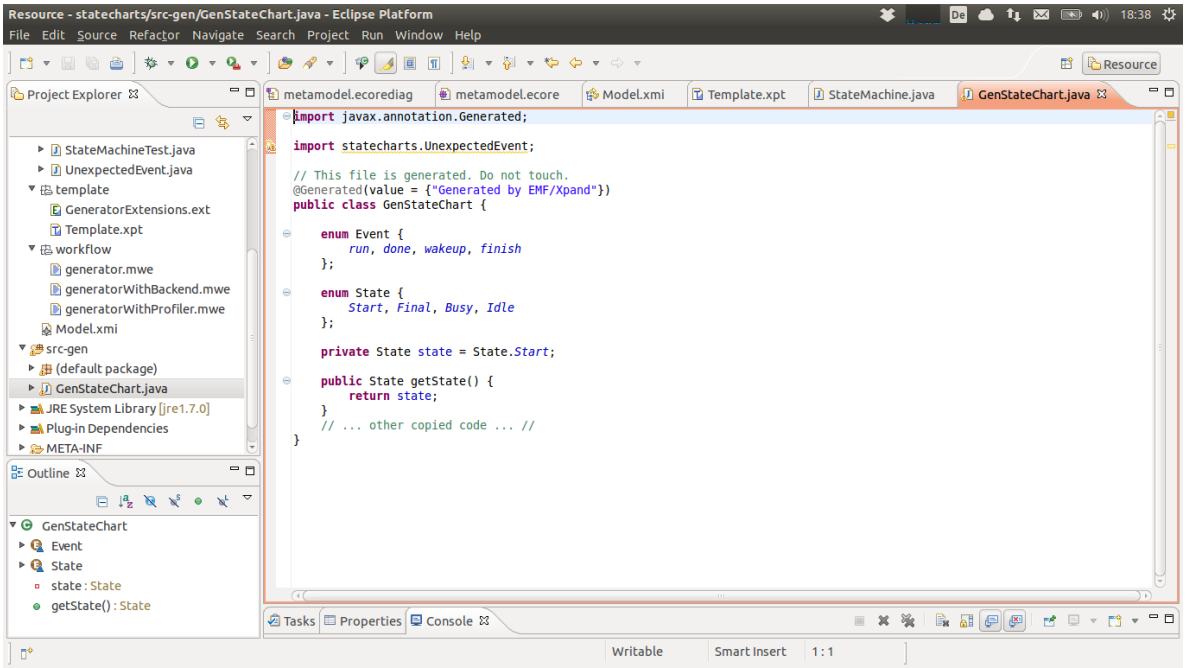
run code generator

Right-click on `generator.mwe`



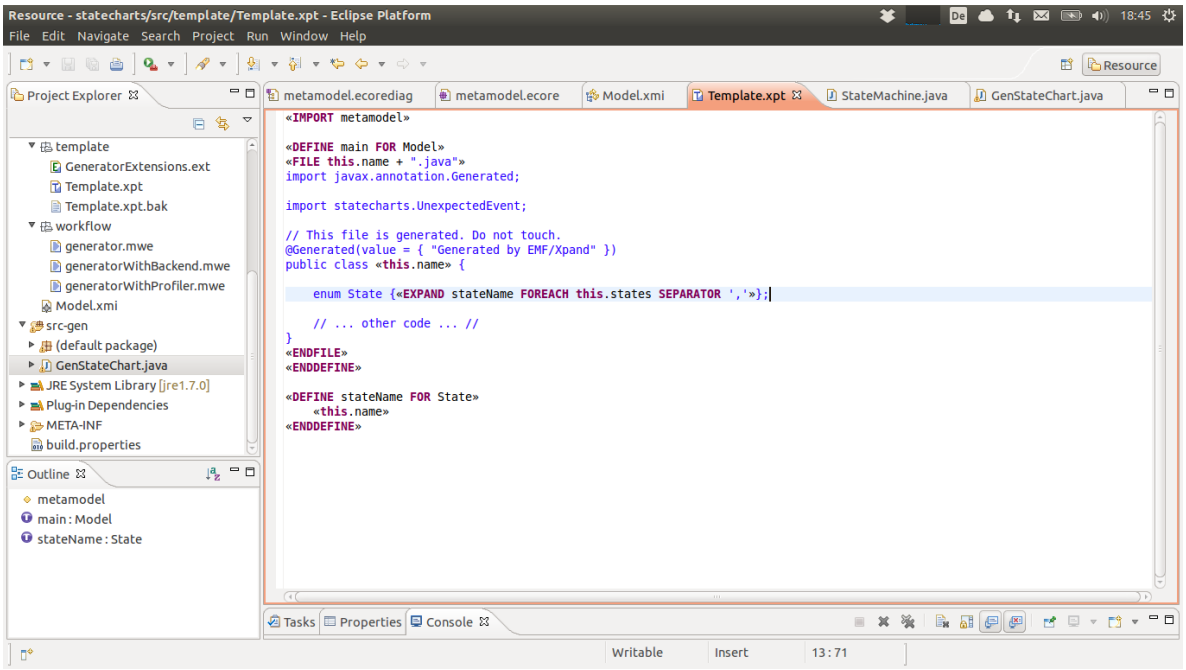
run code generator

Open generated file `GenStateChart.java`



create code generator

Replace hand-written code piece by piece



create code generator

Setze initialen Wert für Variable `state`:

```
private State state = State.Start;
```

Hinweis:

```
attribute.selectFirst(e|e.attr == X)
```

liefert das erste Element des Sequenzattributs `attribute`, das die Eigenschaft `e.attr == X` erfüllt

create code generator

Generiere die `switch-Anweisungen`:

- Makroexpansionen dürfen Parameter haben
- `attribute.select(e|expression e)` liefert Sequenz aller Elemente von `attribute`, die die Eigenschaft `expression e` erfüllen

create code generator I

```
1  ≪ IMPORT metamodel ≫
2
3  ≪ DEFINE main FOR Model ≫
4  ≪ FILE this.name + ".java" ≫
5  import javax.annotation.Generated;
6
7  import statecharts.UnexpectedEvent;
8
9  @Generated(value = { "Generated by EMF/Xpand" })
10 public class ≪this.name≫ {
11
12   enum State {≪ EXPAND stateName
13                 FOREACH this.states SEPARATOR ',' ≫};
14
15   enum Event {≪ EXPAND eventName
16                 FOREACH this.events SEPARATOR ',' ≫};
17
18   private State state =
19         State.≪states.selectFirst(s|s.isInitial).name ≫;
20
```

create code generator II

```
21    public State getState() {
22      return state;
23    }
24
25    public void signal(Event event) throws UnexpectedEvent {
26      switch(state) {
27      « EXPAND state(this) FOREACH states »
28      }
29    }
30
31    private void error() throws UnexpectedEvent {
32      System.err.println("Error");
33      throw new UnexpectedEvent();
34    }
35  }
36  « ENDFILE »
37  « ENDDEFINE »
38
39
40
41  « DEFINE stateName FOR State »
```


create code generator III

```
42    «this.name»
43  « ENDDEFINE »
44
45  « DEFINE eventName FOR Event »
46    «this.name»
47  « ENDDEFINE »
48
49  « DEFINE state(Model model) FOR State »
50    case «this.name»:
51       switch(event) {
52       « EXPAND transition
53        FOREACH model.transitions.select(t|t.from == this) »
54       default: error(); break;
55       }
56       break;
57  « ENDDEFINE »
58
59
60
61  « DEFINE transition FOR Transition »
62    case «guard.name»:
```

create code generator IV

```
63          state = State.≪to.name≫;
64          break;
65  ≪ ENDDEFINE ≫
```


QVT

Query View Transformation (QVT) für Modell-zu-Modell-Transformation:

- Abbildung von einem Modell eines Meta-Modells M zu einem anderen Modell eines Meta-Modells M'
- OMG-Standard
- QVT-Operational: imperative Sprache für unidirektionale Transformationen

`http://help.eclipse.org/juno/topic/org.eclipse.m2m.qvt.oml.doc/references/M2M-QVTO.pdf`

QVT

Beispiel:

- Eingabe: Zustandsmaschine M
- Ausgabe: Zustandsmaschine M', wobei $M \subseteq M'$ und M' enthält explizite Transition $s \xrightarrow{e} s_{error}$ zu einem neuen Fehlerzustand s_{error} für jeden Zustand $s \in M$, falls es in M noch keine solche Transition gibt


```
1  modeltype statechart "strict"
2       uses "http://www.example.org/metamodel";
3
4  // transforms a source statechart into a new target statechart
5  // source where target has all states, transitions, and events
6  // of but extra transitions from every state S to a new 'error'
7  // state for every event for which no outgoing transition at S
8  // exists
9  transformation addErrorState
10    (in source : statechart, out target : statechart);
11
12 // start of transformation
13 main() {
14   //  source.rootObjects() yields all root nodes of source
15   // ! yields exactly one element
16   // [Model] specifies that the instance type must be Model
17   // map is a function application: transform() is applied to
18   // root model instance; transform() is defined below
19   source.rootObjects()![Model].map transform()
20 }
21
22
```

```
23 // maps model
24 mapping Model::transform() : Model {
25   // target model is a subset of source model
26   // => copy all states, transitions, and events of source to
27   //    target model
28   // first map primitive attribute
29   name        := self.name;
30   // then map children onto newly created instances
31   states      := self.states->map copy();
32   events      := self.events->map copy();
33   transitions := self.transitions->map copy();
34
35   // additional elements (error state and explicit transitions)
36   // the new explicit error state
37   var errorState = object State
38         {name := "Error"; isFinal := true; isInitial := false;};
39
40   // add to states
41   states += errorState;
42
43
44
45   // add explicit missing transitions to error state
```



```
46   self.states->forEach(state) {
47     // outgoing transitions of state
48     var outgoingTransitions
49       := self.transitions->select(trans | trans._from = state);
50       // the events of all outgoing transitions
51     var handledEvents
52       := outgoingTransitions->collect(trans | trans.guard);
53     // all events defined in the model
54     var allEvents := self.events;
55     // events not defined for state
56     var unhandledEvents := allEvents - handledEvents->asSet();
57     // create and add new transition to errorState
58     transitions += unhandledEvents->map toErrorTransition
59                                          (state, errorState);
60   };
61 }
62
63 // maps onto equivalent event
64 mapping Event::copy() : Event {
65   name := self.name;
66 }
67
68 // maps onto an equivalent state
```

```
69 mapping State::copy() : State {
70   name      := self.name;
71   isFinal   := self.isFinal;
72   isInitial := self.isInitial;
73 }
74 // maps onto an equivalent transition
75 mapping Transition::copy() : Transition {
76   // resolveone(T) finds corresponding target of type T
77   guard := self.guard.resolveone (Event);
78   // from is a keyword; escape with _
79   _from := self._from.resolveone(State);
80   to    := self.to.resolveone(State);
81 }
82
83
84
85
86
87
88
89 // maps onto a new transition from state to errorState for
90 // given event
91 mapping Event::toErrorTransition
```



```
92     (in state : State, in errorState : State) : Transition
93 {
94   _from := state.resolveone(State);
95   to    := errorState;
96   guard := self.resolveone(Event);
97 }
```

Software-Produktlinien
- Lernziele
- Software-Wiederverwendung
- Erfolgsgeschichten
- Definition
- Übersicht
- Kostenaspekte
- Practice Areas
- Entwicklung der Core-Assets
- Produktentwicklung
- Essentielle Aktivitäten
- Einführung von Produktlinien
- Implementierungsstrategien
- Schwierigkeiten
- Wiederholungsfragen

Samsung's product line:

Lernziele

Software-Produktlinien

- Definition und Bedeutung
- Vor- und Nachteile
- Technische Aspekte
- Organisatorische Aspekte

N.B.: Basiert auf Folien von Linda Northrop
`http://www.sei.cmu.edu/productlines/presentations.html`

Software-Wiederverwendung

1960: Unterprogramme

1970: Module

1980: Objekte

1990: Komponenten

→ opportunistische Wiederverwendung im Kleinen; hat nicht den erwarteten Erfolg gebracht

Software-Produktlinien: geplante Wiederverwendung auf allen Ebene für Familien ähnlicher Systeme

Wiederverwendbares

- Komponenten
- Software-Dokumentation
- Architektur
- Tests (unter anderem Integrations-, Leistungs- und Komponententests)
- Anforderungsspezifikation
- Entwicklungsprozess, Methoden und Werkzeuge
- Budget-/Zeit- und Arbeitspläne
- Handbücher
- Entwickler

Erfolgsgeschichten

CelsiusTech: Familie von 55 Schiffssystemen

- Integrationstest of 1-1,5 Millionen SLOC benötigt 1-2 Leute
- Rehosting auf neue Plattform/Betriebssystem benötigt 3 Monate
- Kosten- und Zeitplan werden eingehalten
- Systemattribute (wie Performanz) können vorausgesagt werden
- hohe Kundenzufriedenheit
- Hardware-/Software-Kostenverhältnis veränderte sich von 35:65 zu 80:20

http://www.sei.cmu.edu/productlines/casestudies/celsiustech/

Erfolgsgeschichten

Nokia: Produktlinie mit 25-30 neuen Produkten pro Jahr.

Produktübergreifend gibt es

- unterschiedliche Anzahlen von Tasten
- unterschiedliche Display-Größen
- andere unterschiedliche Produktfunktionen
- 58 verschiedene unterstützte Sprachen
- 130 bediente Länder
- Kompatibilität mit früheren Produkten
- konfigurierbare Produktfunktionen
- Änderung der Geräte nach Auslieferung

Software-Produktlinie

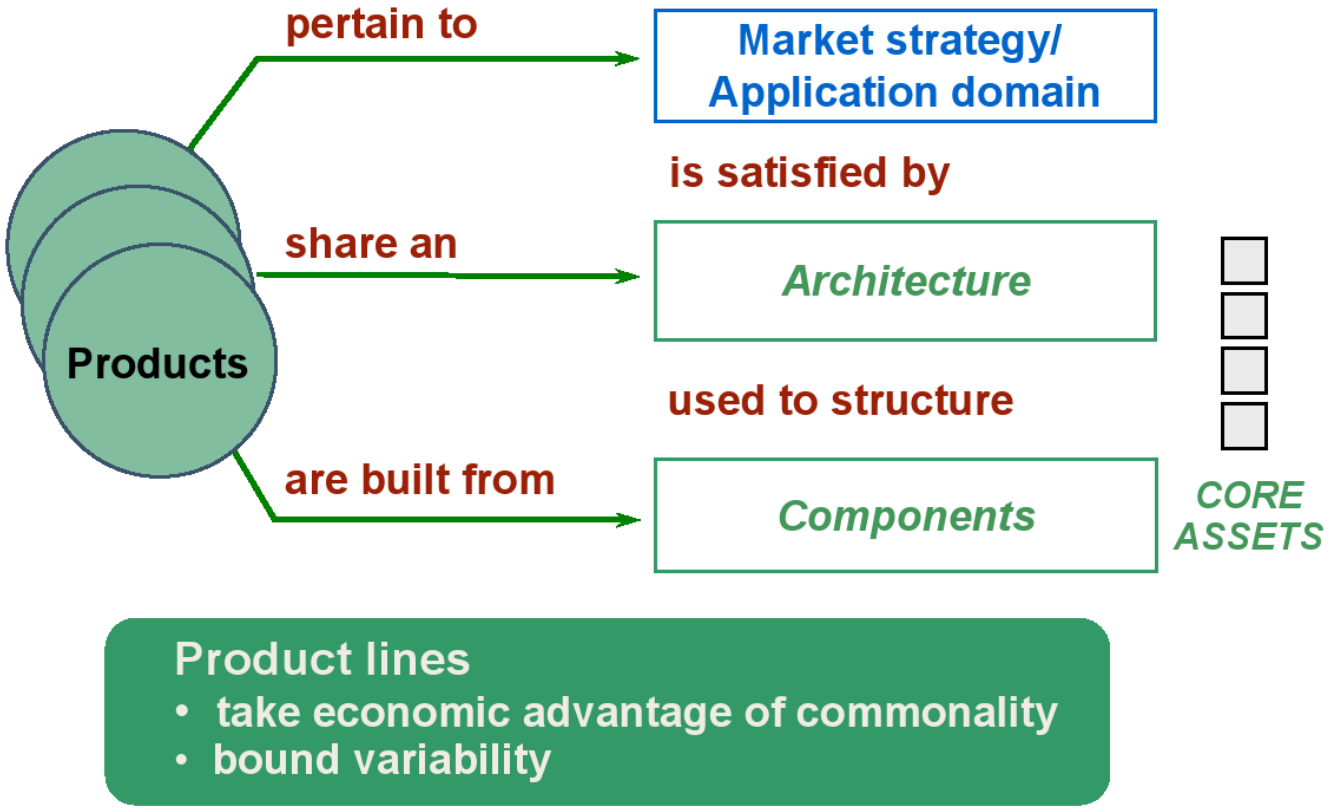
Definition

A **software product line** is a set of software-intensive systems

- sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission
- and that are developed from a common set of core assets in a prescribed way.

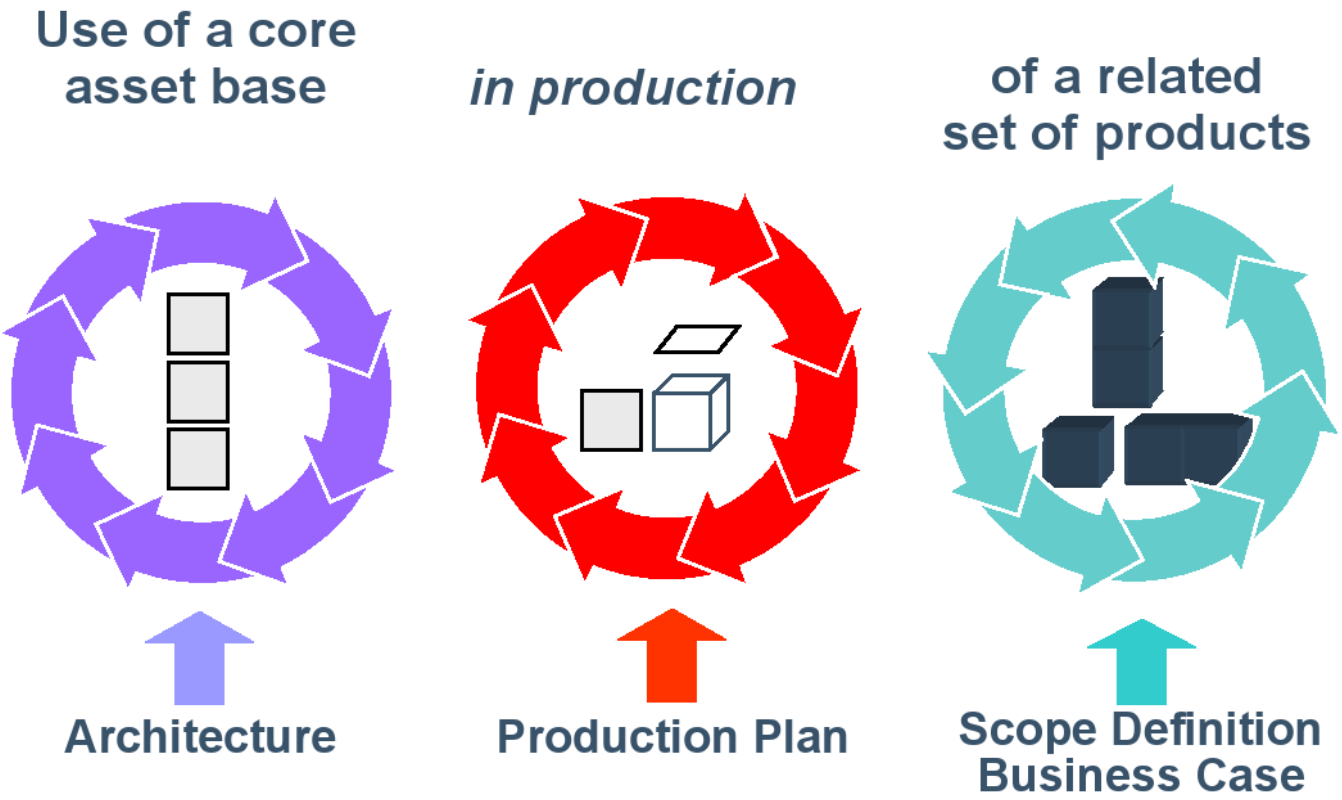
– Clements und Northrop (2001)

Übersicht über Produktlinien



Quelle: Linda Northrop, SEI

Schlüsselkonzepte

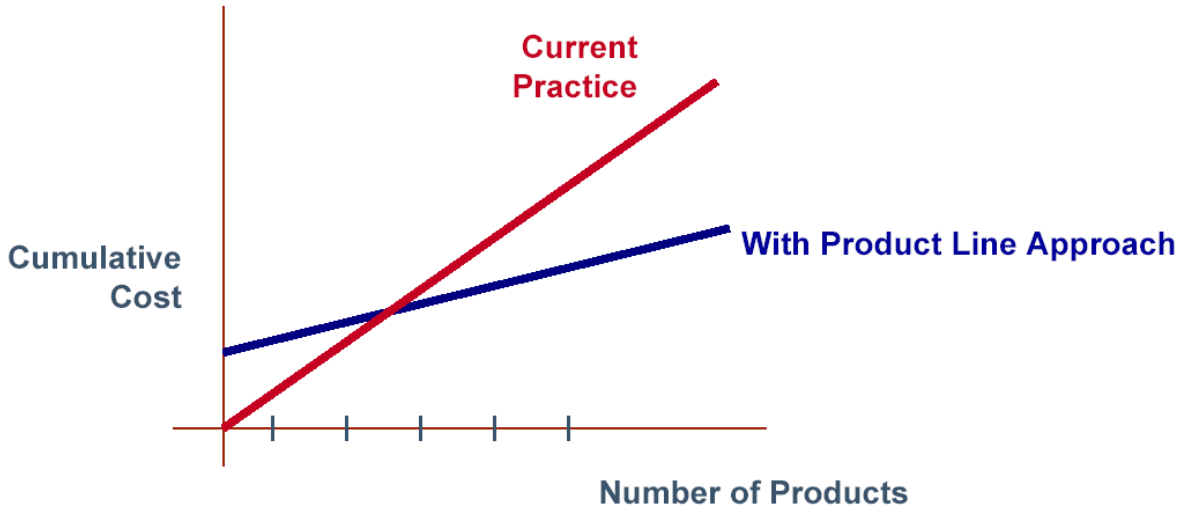


Quelle: Linda Northrop, SEI

Kostenaspekte einer Software-Produktlinie

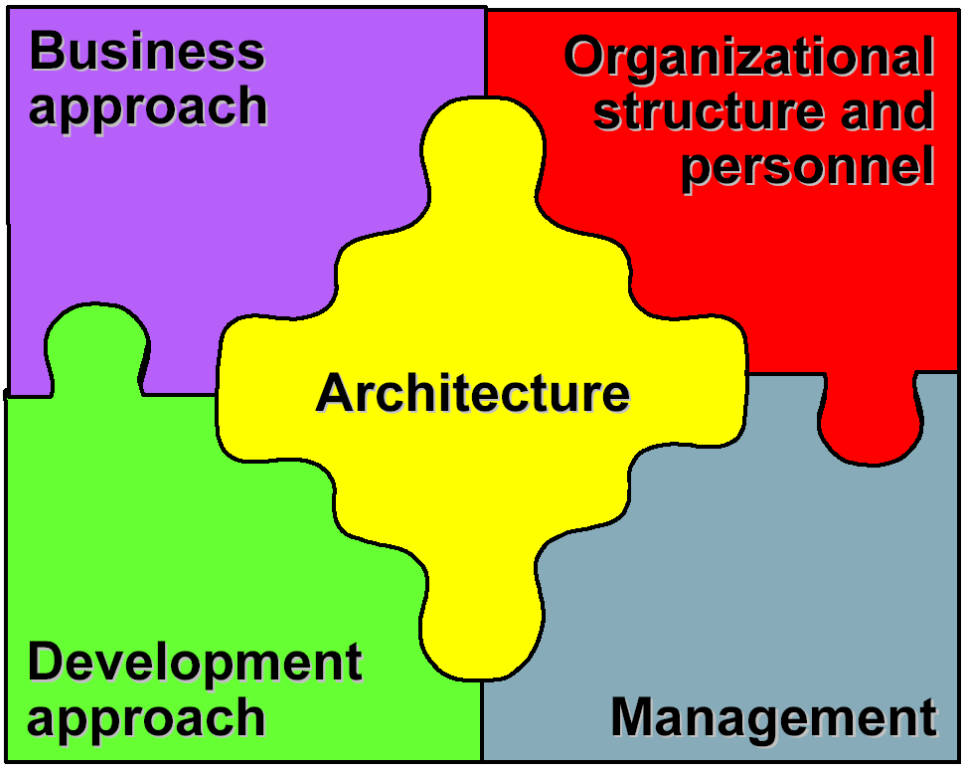
- Marktanalyse: muss eine Familie von Produkten betrachten
- Projektplan: muss generisch oder erweiterbar sein, um Variationen zu erlauben
- Architektur: muss Variation unterstützen
- Software-Komponenten: müssen generischer sein, ohne an Performanz einzubüßen; müssen Variation unterstützen
- Testpläne/-fälle/-daten: müssen Variationen und mehrere Instanzen einer Produktlinie berücksichtigen
- Entwickler: benötigen Training in den Assets und Verfahren der Produktlinie

Return-on-Investment



Quelle: Weiss und Lai, 1999.

Zusammenspielende Komponenten

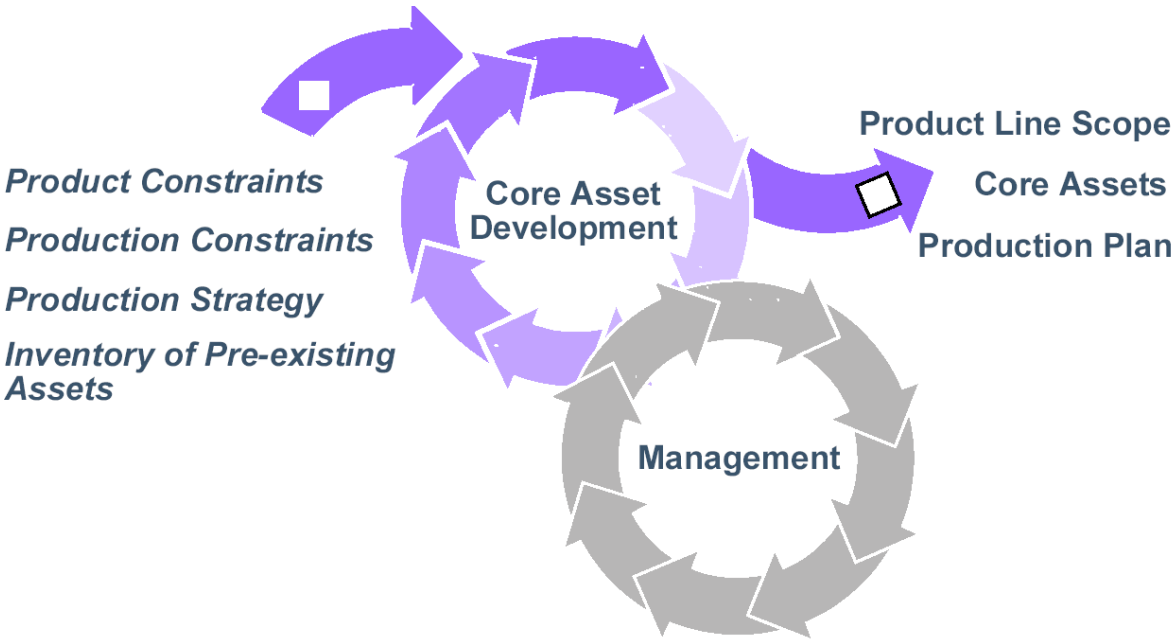


Quelle: Linda Northrop, SEI

Practice Areas

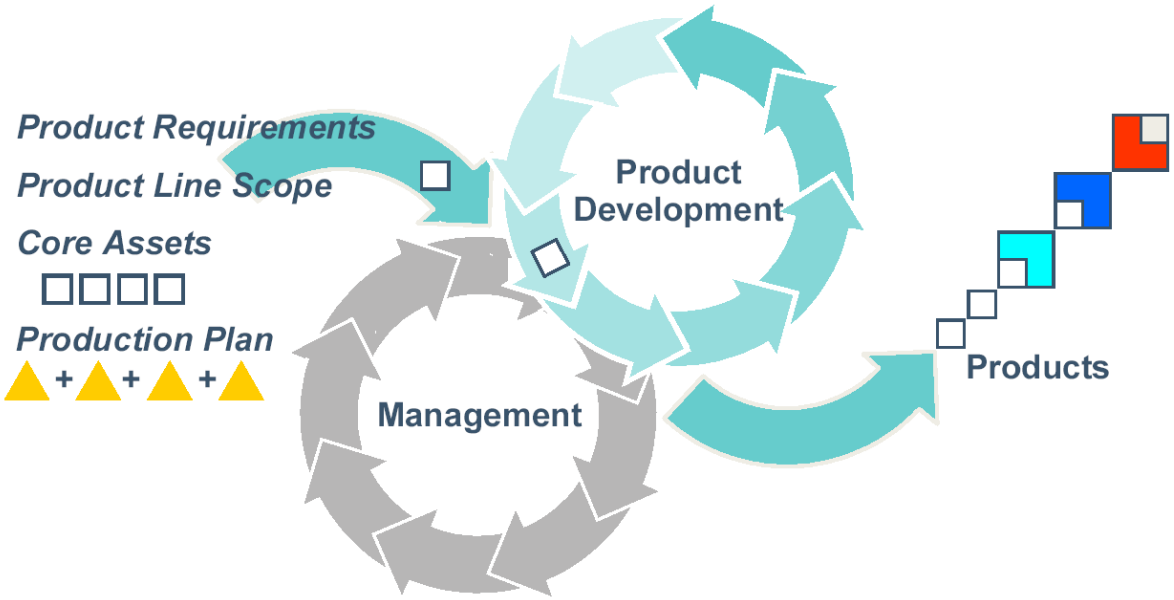
Quelle: Linda Northrop, SEI

Entwicklung der Core-Assets



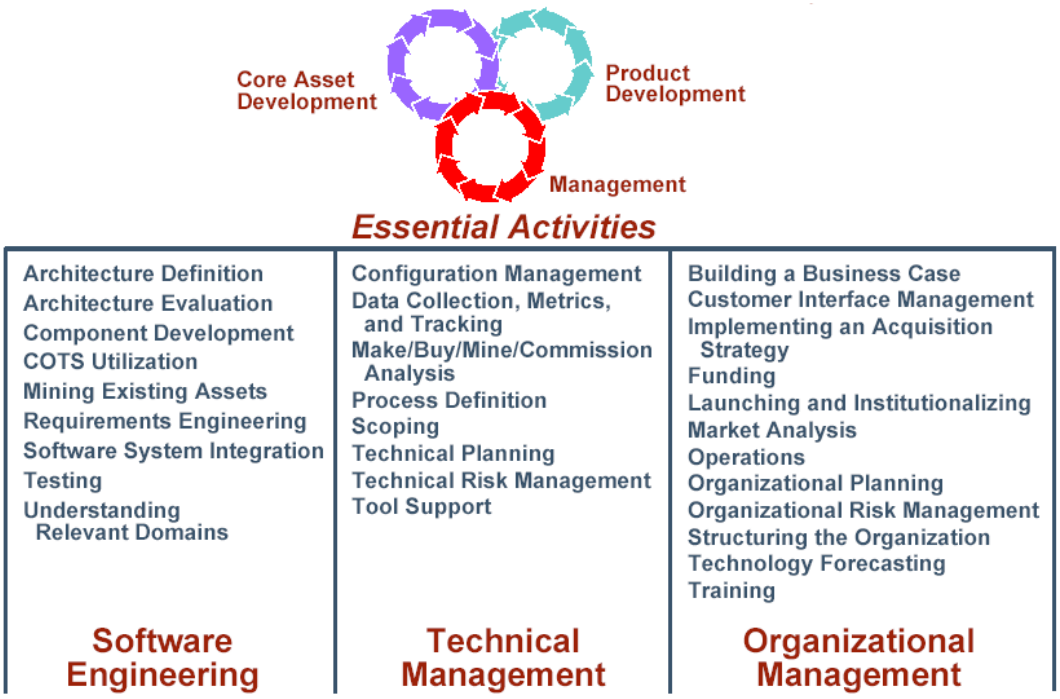
Quelle: Linda Northrop, SEI

Produktentwicklung



Quelle: Linda Northrop, SEI

Essentielle Aktivitäten



Quelle: Linda Northrop, SEI

Einführung von Produktlinien I

Proaktiv:

- definiere zuerst Scope: was gehört zur Produktlinie?
- Scope leitet die weitere Entwicklung
- Entwickle zuerst Core-Assets

\+ Produkte können rasch entwickelt werden, sobald die Produktlinie steht

\- hohe Vorausleistung und Vorhersagefähigkeit verlangt

Einführung von Produktlinien II

Reaktiv:

- Beginne mit einem oder mehreren Produkten
- Extrahiere daraus Core-Assets für die Produktlinie
- Scope entwickelt sich dabei stetig

\+ niedrige Einstiegskosten

\+ größerer Einfluss von Erfahrung

\- Architektur könnte suboptimal sein, wird schrittweise weiterentwickelt

\- Restrukturierungsaufwand notwendig

Einführung von Produktlinien III

Inkrementell (sowohl bei reaktiver als auch proaktiver Entwicklung möglich):

- schrittweise Entwicklung der Core-Assets mit initialer Planung der Produktlinie:
- entwickle Teile der Core-Asset-Base einschließlich Architektur und Komponenten
- entwickle ein oder mehrere Produkte
- entwickle weitere Core-Assets
- entwickle weitere Produkte
- entwickle Core-Asset-Base weiter
- ...

Bindung

Produktlinien ...

- haben Gemeinsamkeiten
- und definierte Unterschiede: Variabilitäten

Produkt wird aus Core-Assets zusammengebaut.
Variabilitäten werden festgelegt.

Bindungszeitpunkt der Variabilitäten

- zur Übersetzungszeit
- zur Bindezeit
- zur Laufzeit

Architekturmechanismen für Variabilitäten

Kombination, Ersetzung und Auslass von Komponenten (auch zur Laufzeit)

Frontend {C, C++, Java}	Middle End {ME}	Backend {i386, Motorola 68000}
C	ME	i386
C	ME	Motorola 68000
C++	ME	i386
C++	ME	Motorola 68000
Java	ME	i386
Java	ME	Motorola 68000

Architekturmechanismen für Variabilitäten

Parametrisierung (einschließlich Makros und Templates)

```
1  generic
2    type My_Type is private;
3    with procedure Foo (M : My_Type);
4  procedure Apply;
5
6  procedure Apply is
7    X : My_Type;
8  begin
9
10     ...
11     Foo (X);
12     ...
13 end Apply;
```

Architekturmechanismen für Variabilitäten

Parametrisierung (einschließlich Makros und Templates)

```
1 typedef (*FP)(int);
2 void Apply (FP fp) {
3     ...
4     fp (X);
5     ...
6 }
```


Architekturmechanismen für Variabilitäten

Selektion verschiedener Implementierungen zur Übersetzungszeit (z.B. #ifdef oder Makefile)

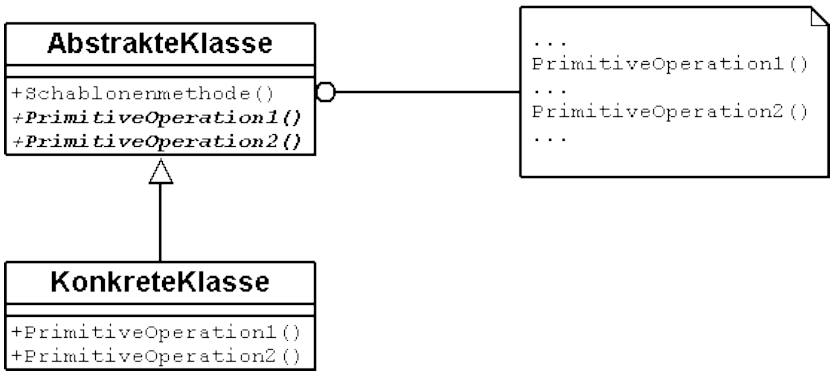
```
1 #ifdef Kunde1
2 #define My_Type int
3  void Foo (My_Type M) {...}
4 #else
5  typedef struct mystruct {...} mystruct;
6 #define My_Type mystruct
7  void Foo (My_Type M) {...}
8 #endif
9  void Apply () {
10   My_Type X;
11     ...
12     Foo (X);
13     ...
14 }
```

Architekturmechanismen für Variabilitäten

Entwurfsmuster mit OO-Techniken

- Strategy
- Factory
- Template Method
- ...

Template Method (Schablonenmethode)



Architekturmechanismen für Variabilitäten

Generierung (z.B. Yacc: Grammatik → Code) und aspektorientierte Programmierung

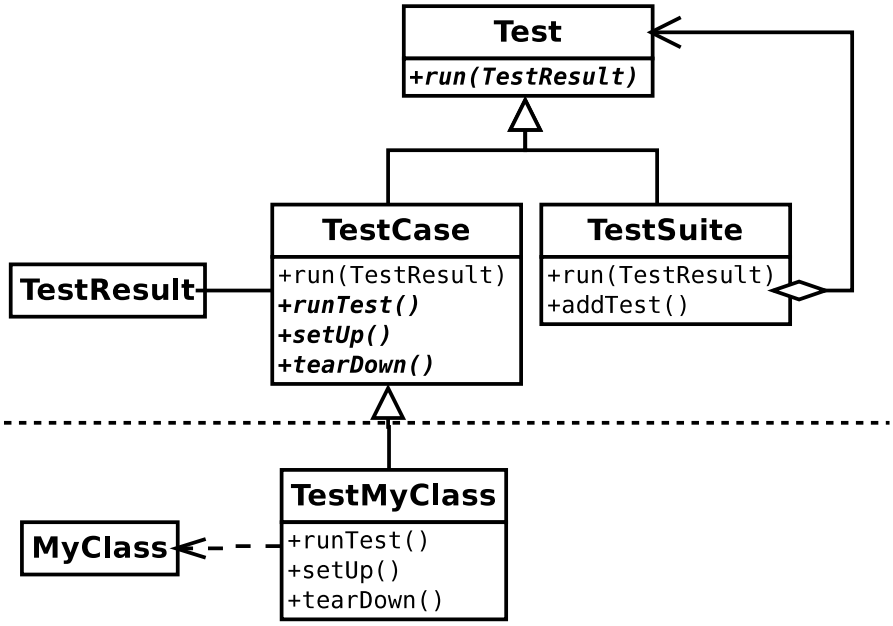
```
1 aspect SetsInRotateCounting {
2     int rotateCount = 0;
3
4     before(): call(void Line.rotate(double)) {
5         rotateCount++;
6     }
7 }
```

- Wie oft wird Methode Line.rotate aufgerufen?

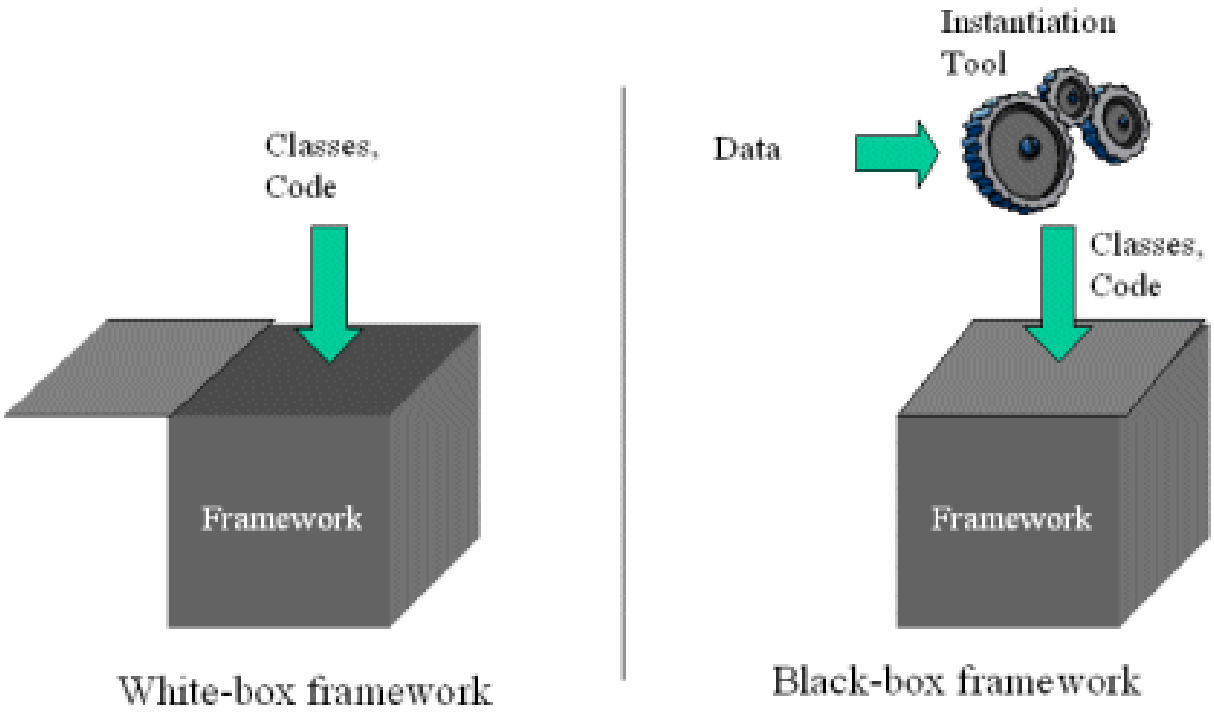
Architekturmechanismen für Variabilitäten

Definition

Anwendungsrahmenwerke (Frameworks): A framework is a set of classes that embodies an abstract design for solutions to a family of related problems. Johnson und Foote (1988)



Anwendungsrahmenwerke (Frameworks)



Schwierigkeiten bei Produktlinien

- Änderung der Organisation (Kern-/Produktaufteilung)
- Was gehört zum Kern?
- Änderungen im Kern haben Auswirkungen auf alle Produkte
- Viele Probleme sind noch nicht gelöst:
  - Test
  - Konfigurationsmanagement
  - ...

Wiederholungs- und Vertiefungsfragen

- Erläutern Sie die Ideen von Software-Produktlinien. Was verspricht man sich davon?
- Was sind die Vor- und Nachteile?
- Wie wird die Entwicklung von Software-Produktlinien organisatorisch häufig strukturiert?
- Erläutern Sie einige essentielle Aktivitäten der Softwareentwicklung und ihre Besonderheiten im Kontext von Software-Produktlinien.
- In welcher Weise können Produktlinien eingeführt werden?
- Beschreiben Sie Implementierungsmechanismen für die Variabilität in Software-Produktlinien. Nennen Sie hierbei den jeweiligen Bindungszeitpunkt. Was drückt der Bindungszeitpunkt aus?

1 **Albrecht 1979** Albrecht, Alan: Measuring application development productivity. In: Proc. Joint SHARE/GUIDE/IBM Applications Development Symposium, 1979, S. 83–92

2 **Balzert 1997** Balzert, Helmut: Lehrbuch der Software-Technik. Spektrum Akademischer Verlag, 1997. – ISBN 3827400651

3 **Balzert 2008** Balzert, Helmut: Lehrbuch der Softwaretechnik – Softwaremanagement. 2. Spektrum, Akademischer Verlag, 2008. – ISBN 978-3-8274-1161-7

4 **Banker u. a. 1991** Banker, R. ; Kauffmann, R. ; Kumar, R.: An Empirical Test of Object-Based Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment. In: Journal of Management Information Systems 8 (1991), Nr. 3, S. 127–150

5 **Basili und Weiss 1984** Basili, R. ; Weiss, D. M.: A Methodology for Collecting Valid Software Engineering Data. In: IEEE Transactions on Software Engineering 10 (1984), November, Nr. 6, S. 728–738

6 **Basili und Turner 1975** Basili, V. ; Turner, J.: Iterative Enhancement: A Practical Technique for Software Development. In: IEEE Transactions on Software Engineering 1 (1975), Dezember, Nr. 4, S. 390–396

7 **Bass u. a. 2003** Bass, Len ; Clements, Paul ; Kazman, Rick: Software Architecture in Practice. 2nd ed. Addison Wesley, 2003

8 **Beck 2000** Beck, Kent: Extreme Programming Explained. Addison-Wesley, 2000 (The XP Series). – ISBN 201-61641-6

9 **Boehm 1981** Boehm, B.: Software Engineering Economics. Prentice Hall, 1981

10 **Boehm und Turner 2003** Boehm, B. ; Turner, R.: Using risk to balance agile and plan-driven methods. In: IEEE Computer 36 (2003), Juni, Nr. 6, S. 57–66

11 Boehm 1979 Boehm, Barry W.: Guidelines for verifying and validating software requirements and design specification. In: EURO IFIP 79, 1979, S. 711–719

12 Boehm 1988 Boehm, Barry W.: A spiral model of software development and enhancement. In: IEEE Computer 21 (1988), Mai, Nr. 5, S. 61–72

13 Boehm u. a. 2000 Boehm, Barry W. ; Abts, Chris ; Brown, A. W. ; Chulani, Sunita ; Clark, Bradford K. ; Horowitz, Ellis ; Madachy, Ray ; Reifer, Donald ; Steece, Bert: Software Cost Estimation with COCOMO II. Prentice Hall, 2000

14 Bortz und Döring 2006 Bortz, Jürgen ; Döring, Nicloa: Forschungsmethoden und Evaluation. vierte Auflage. Springer, 2006. – ISBN 978-3-540-33305-0

15 Bortz u. a. 2008 Bortz, Jürgen ; Lienert, Gustav A. ; Böhnke, Klaus: Verteilungsfreie Methoden in der Biostatistik. zweite Ausgabe. Springer Verlag, 2008. – ISBN 3540675906

16 Brooks 1995 Brooks, Frederick P.: The Mythical Man-Month: Essays on Software Engineering, Anniversary Edit 2. Addison-Wesley Professional, 1995. – ISBN 0201835959

17 Bunse und von Knethen 2002 Bunse, Christian ; Knethen, Antje von: Vorgehensmodelle kompakt. Spektrum-Akademischer Verlag, 2002. – ISBN 978-3827412034

18 Buschmann u. a. 1996 Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter ; Stal, Michael: Pattern-oriented Software Architecture: A System of Patterns. Bd. 1. Wiley, 1996

19 Chidamber und Kemerer 1994 Chidamber, S.R. ; Kemerer, C.F.: A Metrics Suite for Object Oriented Design. In: IEEE Transactions on Software Engineering 20 (1994), Nr. 6, S. 476–493

20 Christensen 2007 Christensen, Larry B.: Experimental Methodology. 10th edition. Pearson International Edition, 2007. – ISBN 0-205-48473-5

21 Clements und Northrop 2001 Clements, Paul ; Northrop, Linda M.: Software Product Lines : Practices and Patterns. Addison Wesley, August 2001. – ISBN 0201703327

22 Cobb und Mills 1990 Cobb, R. H. ; Mills, H. D.: Engineering Software Under Statistical Quality Control. In: IEEE Software 7 (1990), Nr. 6, S. 44–54

23 Dzidek u. a. 2008 Dzidek, Wojciech J. ; Arisholm, Erik ; Briand, Lionel C.: A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. In: IEEE Transactions on Software Engineering 34 (2008), May/June, Nr. 3

24 Endres und Rombach 2003 Endres, Albert ; Rombach, Dieter: A Handbook of Software and Systems Engineering. Addison Wesley, 2003

25 Favre 2004 Favre, J. M.: Foundations of Meta-Pyramids: Languages vs. Metamodels - Episode II: Story of Thotus the Baboon. In: Bézivin, J. (Hrsg.) ; Heckel, R. (Hrsg.): Language Engineering for Model-Driven Software Development Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (Veranst.), 2004

26 Fenton und Pfleeger 1998 Fenton, N. ; Pfleeger, S.: Software Metrics: A Rigorous & Practical Approach. 2nd. London : International Thomson Computer Press, 1998

27 Gamma u. a. 2003 Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: Design Patterns—Elements of Reusable Object-Oriented Software. Addison Wesley, 2003

28 Garlan u. a. 1995 Garlan, D. ; Allen, R. ; Ockerbloom, J.: Architectural Mismatch: Why Reuse is So Hard. In: IEEE Software 12 (1995), November, Nr. 6, S. 17–26

29 Gilb 1988 Gilb, Tom: Principles of Software Engineering Management. Harlow, UK : Addison-Wesley, 1988

30 Gornik 2001 Gornik, David: IBM Rational Unified Process: Best Practices for Software Development Teams / IBM Rational Software. 2001 (TP026B, Rev 11/01). – White Paper

31 Halstead 1977 Halstead, Maurice H.: Elements of Software Science. In: Operating, and Programming Systems Series 7 (1977)

32 Hofmeister u. a. 2000 Hofmeister, Christine ; Nord, Robert ; Soni, Dilip: Applied Software Architecture. Addison Wesley, 2000 (Object Technology Series)

33 Humphrey 1995 Humphrey, Watts S.: A Discipline For Software Engineering. Addison-Wesley, 1995 (SEI series in software engineering)

34 IBM 1985 : Die Function-Point-Methode: Eine Schätzmethode für IS-Anwendungsprojekte. IBM-Form GE12-1618-1. 1985

35 Jones 1995 Jones, Capers: Backfiring: Converting Lines of Code to Function Points. In: IEEE Computer 28 (1995), November, Nr. 11, S. 87–88

36 Jones 1996 Jones, Capers: Software Estimating Rules of Thumb. In: IEEE Computer 29 (1996), March, Nr. 3, S. 116–118

37 Kauffman und Kumar 1993 Kauffman, R. ; Kumar, R.: Modeling Estimation Expertise in Object Based ICASE Environments / Stern School of Business, New York University. Januar 1993. – Report

38 Kemerer 1987 Kemerer, Chris F.: An Empirical Validation of Software Cost Estimation Models. In: Comm. ACM 30 (1987), May, Nr. 5

39 Kemerer und Porter 1992 Kemerer, Chris F. ; Porter, Benjamin S.: Improving the Reliability of Function Point Measurement: An Empirical Study. In: TSE 18 (1992), Nov., Nr. 11

40 Kleppe 2008 Kleppe, A: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley, Pearson Education Inc., 2008

41 Kneuper 2006 Kneuper, Ralf: CMMI – Verbesserung von Softwareprozessen mit Capability Maturity Model. 2. dpunkt.verlag, 2006. – ISBN 3-89864-373-5

42 Knight und Leveson 1986 Knight, J.C. ; Leveson, N.G.: An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. In: IEEE Transactions on Software Engineering 12 (1986), Januar, Nr. 1, S. 96–109

43 Kruchten 1998 Kruchten, Phillipe: The Rational Unified Process: An Introduction. Reading, Mass.: Addison-Wesley, 1998

44 Lienert 1973 Lienert, G.A.: Verteilungsfreie Methoden in der Biostatistik. Meisenheim am Glan, Germany : Verlag Anton Hain, 1973. – wird leider nicht mehr aufgelegt

45 Ludewig und Lichter 2006 Ludewig, Jochen ; Lichter, Horst: Software Engineering – Grundlagen, Menschen, Prozesse, Techniken. dpunkt.verlag, 2006

46 McCabe 1976 McCabe, T.: A Software Complexity Measure. In: IEEE Transactions on Software Engineering 2 (1976), Nr. 4, S. 308–320

47 Mernik u. a. 2005 Mernik, M. ; Heering, J. ; Sloane, A. M.: When and how to develop domain-specific languages. In: ACM Computing Surveys 37 (2005), Nr. 4, S. 316–344

48 Mills u. a. 1987 Mills, H.D. ; Dyer, M. ; Linger, R.: Cleanroom Software Engineering. In: IEEE Software 4 (1987), September, Nr. 5, S. 19–25

49 Moore u. a. 2009 Moore, David S. ; McCabe, George P. ; Craig, Bruce A.: Introduction to the Practice of Statistics. sixth edition. W.H. Freeman and Company, 2009

50 Müller 2006 Müller, Matthias M.: Do Programmer Pairs make different Mistakes than Solo Programmers? In: Conference on Empirical Assessment In Software Engineering, April 2006

51 OSGI Alliance OSGI Alliance: OSGI. – URL `http://www.osgi.org/Main/HomePage`

52 Parker 1995 Parker, Burton G.: Data Management Maturity Model. July 1995

53 Pichler 2008 Pichler, Roman: Scrum — Agiles Projektmanagement erfolgreich einsetzen. dpunkt.verlag, 2008. – ISBN 978-3-89864-478-5

54 Poensgen und Bock 2005 Poensgen, Benjamin ; Bock, Bertram: Die Function-Point-Analyse. Ein Praxishandbuch. Dpunkt Verlag, 2005. – ISBN 978-3898643320

55 Prechelt 2001 Prechelt, Lutz: Kontrollierte Experimente in der Softwaretechnik – Potenzial und Methodik. Springer, 2001

56 Pressman 1997 Pressman, Roger: Software Engineering – A Practioner's Approach. Vierte Ausgabe. McGraw-Hill, 1997

57 Reussner und Hasselbring 2009 Reussner, R. (Hrsg.) ; Hasselbring, W (Hrsg.): Handbuch der Software-Architektur. zweite Ausgabe. dpunkt.verlag, 2009

58 Royce 1970 Royce, W.: Managing the Development of Large Software Systems. In: Proceedings Westcon, IEEEPress, 1970, S. 328–339

59 Schmidt u. a. 2000 Schmidt, Douglas ; Stal, Michael ; Rohnert, Hans ; Buschmann, Frank: Pattern-oriented Software Architecture: Patterns for Concurrent and Networked Objects. Bd. 2. Wiley, 2000

60 Siviy u. a. 2007 Siviy, Jeannine M. ; Penn, M. L. ; Stoddard, Robert W.: CMMI and Six Sigma – Partners in Process Improvement. Addison-Wesley, 2007 (SEI Series in Software Engineering). – ISBN 978-0-321-51608-4

61 Sneed 2004 Sneed, Harry: Vorlesungsskriptum Software-Engineering. Uni Regensburg: Wirtschaftsinformatik (Veranst.), 2004

62 Sommerville 2004 Sommerville, Ian: Software Engineering. Addison-Wesley, 2004

63 Stachowiak 1973 Stachowiak, H: Allgemeine Modelltheorie. Springer, 1973

64 Stahl u. a. 2007 Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: Modellgetriebene Softwareentwicklung – Techniken, Engineering, Management. zweite Auflage. dpunkt.verlag, 2007

65 Symons 1988 Symons, C. R.: Function Point Analysis: Difficulties and Improvements. In: TSE 14 (1988), Jan., Nr. 1, S. 2–11

66 Szyperski u. a. 2002 Szyperski, Clemens ; Gruntz, Dominik ; Murer, Stephan: Component Software. Second edition. Addison-Wesley, 2002. – ISBN 0-201-74572-0

67 Tichy 1998 Tichy, Walter: Should computer scientists experiment more? In: IEEE Computer 31 (1998), Mai, Nr. 5, S. 32–40

68 Winner u. a. 1991 Winner, B.J. ; Brown, Donald R. ; Michels, Kenneth M.: Statistical Principles in Experimental Design. 3rd edition. McGraw-Hill, 1991 (Series in Psychology)

69 Wohlin u. a. 2000 Wohlin, Claes ; Runeson, Per ; Magnus C. Ohlsson, Martin H. und ; Regnell, Björn ; Wesslén, Anders: Experimentation in Software Engineering – An Introduction. Kluwer Academic Publishers, 2000. – ISBN 0-7923-8682-5

70 Yin 2003 Yin, Robert K.: Applied Social Research Methods Series. Bd. 5: Case Study Research. 3rd edition. SAGE Publications, 2003. – ISBN 0-7619-2553-8